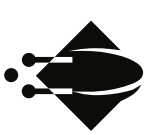
Incremental Interactive Verification of the Correctness of Object-Oriented Software

Smash the State, One Field at a Time

Hannes Mehnert



IT University
of Copenhagen

A dissertation submitted to the PhD school at the IT University of Copenhagen for the degree of Doctor of Philosophy.

Submitted: July 2013, Copenhagen
Defended: 11th October 2013, Copenhagen
Final version: December 2013, Copenhagen

Advisors:

Peter Sestoft, IT University of Copenhagen, Denmark
Lars Birkedal, Aarhus University, Denmark

Evaluation committee:

Joseph Roland Kiniry, Technical University of Denmark, Denmark
Marieke Huisman, University of Twente, the Netherlands
Bart Jacobs, KU Leuven, Belgium

Acknowledgements

I want to thank my splendid advisor Peter Sestoft, without whom I would not have started this PhD in the first place. I met Peter during a summer school on Lipari in 2007, where he lectured about programming languages. I was fascinated by the attitude and openness of Peter, characteristics of a professor I have not experienced previously. Peter supported me fully in all possible areas during my PhD.

I also want to thank Lars Birkedal, my second advisor. Lars kept on pushing me by asking the questions which motivated me to think more in the right direction. Both my advisors' offices were always open for conversations, and I always felt welcome to ask any questions I had in mind.

My parents Annette and Michael supported me completely during my studies, for which I am very grateful. My beloved brother Jan also supported and motivated me whenever we met.

There are numerous people who motivated me to do this PhD. A major event to get me started was the ICFP conference in 2005 in Tallinn, Estonia. I was invited to be there due to the programming contest, where my team, the Dylan Hackers, won the judge's prize and also the second prize. At ICFP I met a lot of wonderful people, I will always keep Bob Harper in mind, who approached us after the award ceremony with "You cannot argue against success". The contest was organized by the PLT Scheme group, especially Robby Findler motivated me at the conferences where I met him.

Without the comments and suggestions regarding the design from Anke Riemer, this dissertation would not be as beautiful as it is. I am very thankful for her valuable suggestions.

I appreciate all my proofreaders who spent hours to give me feedback on this dissertation. This includes Jan Ludewig, Peter Sestoft, Lars

Birkedal, Jan Mehnert, David Christiansen, Helge Pfeiffer, Paolo Tell, Alec Faithfull, Bastian Müller, and Jesper Bengtson.

I want to thank researchers whom I met over the years, especially Ronald Garcia, Neel Krishnaswami, Edwin Brady, Jan Midtgaard, and Alex Potanin.

I stayed with Jonathan Aldrich's group at CMU in Pittsburgh for 6 months. I want to thank Jonathan, and the whole PLAID group, especialy Robert Bocchino, Ligia Nistor, Cyrus Omar, Karl Naden, Kevin Bierhoff, and Ciera Jaspan.

This dissertation started in a research project, where I have learned a lot from my colleagues Jonas Jensen, Jakob Thamsborg, Jesper Bengtson, Filip Sieczkowski, and Kasper Svendsen. The grant for the research project 09-065888 "Tools and Methods for Scalable Software Verification" from the Danish Research Council for Technology and Production (DFF-FTP). The discussions in the project meetings were always fruitful.

I'm especially thankful to my colleagues Andrea, Maxime, Josu, Paolo, Helge, and David for fruitful discussions. I want to thank also my fellow PhD students and colleagues at ITU: Christina Neumayer, Elena Nazzi, Francesco Zanitti, Gian Perrone, Paolo Tell, Rosalba Giuffrida, Paolo Burelli, Dario Pacino, Alberto Delgado Ortegon, Josu Martinez, Maxime Beauquier, Jasmine Duchon, Nicolas Pouillard, Daniel Gustafsson, Andrea Campagna, Alec Faithfull, David Christiansen, Helge Pfeiffer, Kostas Pantazos, Ornela Dardha, Joe Kiniry, and Kasper Østerbye.

Furthermore I want to thank all my current and former flatmates, especially Johan Kjær Thillemann, who accomodated me selflessly when I arrived in Copenhagen. I thank My other lovely flatmates Andreas Rasmussen, Vidir Valberg Guðmundsson, Aslak Ransby, Philip Lavender, and Seraina Nett. My deepest thanks go to Ishtar for all the wonderful espresso she brewed.

I want to thank everyone whom I forgot to mention, especially my good friends in Berlin and at lots of places in the world (just to name a few: Wellington, New York, Vadestedet, Montreal, Malmö, San Francisco, Hamburg, Boston, Zagreb, Lärz, Split, Rome, Vitoria-Gasteiz).

I want to thank my evaluation committee, whose feedback was very helpful and improved this dissertation.

Last but not least I want to thank emacs, my favorite operating system (besides FreeBSD) and development environment in which I spend a lot of time since more than 10 years.

Abstract

Development of correct object-oriented software is difficult, in particular if a formalised proof of its correctness is demanded.

A lot of current software is developed using the object-oriented programming paradigm. This paradigm compensated for safety and security issues with imperative programming, such as manual memory management. Popularly used integrated development environments (IDEs) provide features such as debugging and unit testing to facilitate development of robust software, but hardly any development environment supports the development of provable correct software.

A tight integration of a proof assistant into a widely used IDE lowers the burden for a developer to prove the correctness of her software.

This dissertation introduces *Kopitiam*, a plugin for the industry-grade IDE Eclipse to interactively prove the correctness of Java software using separation logic. Kopitiam extends Eclipse's Java development perspective with specifications and proof script annotations, and provides an Eclipse development environment for the well-known interactive proof assistant Coq. Kopitiam does not need to be trusted, because the validity of the constructed correctness proof is checked by Coq. In this dissertation I describe the requirements of Kopitiam and solutions to implementation challenges by presenting several generations of Kopitiam. Kopitiam's usefulness is evaluated qualitatively in an experiment.

I also present a formalised correctness proof of the snapshotable tree data structure. For efficiency, our implementation uses *copy-on-write* and shared mutable data, not observable by a client. I further use this data structure to verify the correctness of a solution to the point location problem. The results demonstrate that I am able to verify the correctness of object-oriented software which is used in the wild.

Contents

List of Figures

List of Tables

Part I

Set and Setting

Chapter 1

Introduction

This dissertation is about verifying the correctness of object-oriented software. First, I will define the meaning of these words.

The Oxford English Dictionary[1] (OED) defines the verb *verify* as *make sure or demonstrate that (something) is true, accurate, or justified*. It originates from the Latin verb *vērificāre*, according to Wiktionary[2]. The verb *vērificāre* was first attested circa 1250. Its etymology is *vērus* (*true*) and *fācīō* (*do, make*). The meaning is *confirm the truth*.

When we apply *verification* to correctness of software, there is no obvious truth. The OED defines software as a *program used by a computer*, and a program as a *series of statements to control the operation of a computer*.

What I actually mean by verifying the correctness of software is that I specify the desired behaviour, model the execution of the software, and demonstrate that the execution of the software corresponds to the specified behaviour. Verifying the correctness of software boils down to convincing a reader that the model is precise, that the specification expresses the desired requirements, and that the proof is correct.

Human fallibility makes it difficult to trust the correctness of a proof. An interactive proof assistant is a program that implements formal logic, and aids in developing and checking mathematical proofs. If the interactive proof assistant accepts a proof and the reader trusts the interactive proof assistant, then the reader only needs to be convinced that the specification and the model are correct.

[1] `http://oxforddictionaries.com/definition/english/verify`
[2] `https://en.wiktionary.org/wiki/verifico`

1.1 Motivation

Most contemporary software is buggy. Who has not encountered a crashing operating system?

Furthermore, there have been several software bugs with lethal consequences. In the 1980s the software controlling the radiation therapy machine Therac-25 caused overdoses of radiation [76] to patients by selecting an incorrect operating mode. More recently, a similar overexposure of radiation therapy patients [25] has also been caused by faulty software. In a survey article Wong et al. [118] describe several catastrophic accidents and in which way software was responsible for them.

It is not my goal to scare the reader away and to live in a cave like in the Stone Age. Rather, I emphasize that software verification is a useful research area in the contemporary world. In recent decades, a lot of progress was made, and there were many successful verification projects. To assess a verification project, we have to bear in mind both the statement that was proven and the verification tool that was used, which is also software that might contain bugs. In this dissertation I mainly use the interactive proof assistant Coq, that is considered to be a very practical and rigorous form of proving currently available due to its small trusted kernel and wide use.

In contrast to verification, software testing alone is not sufficient for equipment that is essential for survival. Already more than 40 years ago, Dijkstra remarked that "*Testing shows the presence, not the absence of bugs*" [38]. Testing yields witnesses that software works correctly for particular inputs, whereas verification provides a universally quantified correctness statement. The topic of this dissertation is *how to verify the correctness of object-oriented programs*.

I focus on object-oriented software, because a large amount of contemporary software is implemented using the object-oriented paradigm. The object-oriented paradigm extends imperative programming with encapsulation. An imperative program consists of a series of statements that *modify* the program *state*. Java is a popular object-oriented programming language, of which we formalised a subset for this dissertation.

A challenging property for the verification of correctness of object-oriented software is to deal with mutable state. The discovery of separation logic [100] about a decade ago enables us to reason modularly about software with mutable state.

Another aspect of software development is *maintenance*. Software is a dynamic system, as Lehman [73] described over 40 years ago. Software evolves over time to comply with changing requirements. It is also refactored, which means that semantics-preserving transformations are applied for maintainability. Brooks [28] also points out that software maintenance is an underestimated cost.

Most object-oriented software is developed with the help of an integrated development environment, which provides several supporting tools for the programmer, such as navigating, documenting, refactoring, debugging, profiling, unit testing, and execution of software.

This dissertation presents *Kopitiam*, which integrates the well-known proof assistant Coq tightly into the industry-grade integrated development environment Eclipse. I use an embedding of higher-order separation logic into Coq to verify the correctness of object-oriented software. By tight integration into Eclipse I enable developers to maintain their proofs.

1.2 Thesis

The thesis of this dissertation is that modular interactive verification of correctness of object-oriented software is achievable and can be tightly integrated into the software development workflow. I have a high faith in the validity of the correctness proofs for a reason: these proofs are formalised (mechanically checkable) in the well-known proof assistant Coq.

1.3 Research Questions

The thesis gives rise to several research questions:

1. Can a proof assistant, in particular Coq, be used to verify concrete programs, in addition to the verification of meta-theoretical properties of a programming language?

2. Are off-the-shelf proof assistants mature enough to cope with the size of proofs of real object-oriented software?

3. Are object-oriented programs verifiable using separation logic, especially if they use shared mutable state not observable by a client?

4. Can a verified program be re-used, or does the correctness proof need to be developed from scratch each time the program is modified and extended?

5. Can a verification tool be integrated into a software development environment such that it is usable by developers?

6. What are the desired properties, forming the design space, of verification tools? How populated is the design space?

These research questions are answered throughout this dissertation. More specifically, research question 1 is answered by our development of Charge! [11, 12]. To answer research question 2 I conducted case study which verifies an implementation of snapshotable trees in Chapter 10. The same chapter also answers research question 3, with an application of the data structure in Chapter 11. By re-using the verification of the snapshotable tree data structure in the proof of the planar point location problem in Chapter 11, I answer research question 4. The usability of verification tools is answered throughout Part II, including design and implementation challenges, and a preliminary user evaluation in Chapter 9. Research question 6 is answered by my feature analysis in Chapter 2.

1.4 Contributions

In Part I (Chapters 1–5) I describe the background and environment of this dissertation. In Chapter 1 I introduce this dissertation. Related work of this dissertation is split into two chapters: In Chapter 2 I present a feature analysis [61] of verification tools, and in Chapter 3 I relate to other research fields. In Chapter 4 I describe the future work and in Chapter 5 I conclude this dissertation.

The contribution of this dissertation is twofold: Firstly in Part II (Chapters 6–9), I present *Kopitiam*, an plugin for the integrated environment Eclipse that tightly integrates Java development and proofs of correctness in Coq.

Secondly in Part III (Chapters 10–12), I show a functional correctness proof of a solution to the *point location problem*. This problem resides in geometry: find for a queried point the smallest containing region, when given a subdivision of a plane into regions upfront.

1.4.1 Kopitiam

I present an initial implementation of Kopitiam in Chapter 6. A more mature generation of Kopitiam and a workflow integrating program development with correctness proofs are described in Chapter 7. I describe the evolutionary design and implementation of four generations of Kopitiam in Chapter 8 by showing the requirements and the intended workflow for a software developer. Chapter 9 presents a pilot empirical evaluation of the usability of Kopitiam by using the technology acceptance model [40, 71].

1.4.2 Point Location

There are several solutions for the point location problem [41, Chapter 6] that use space linear in the number of line segments and provide a logarithmic query time. The planar subdivision is given by a set of non-overlapping line segments which form closed regions. My presented solution [101] meets these time and space boundaries by using the snapshotable tree data structure [46].

The snapshotable tree data structure is a standard binary tree augmented with a snapshot operation, that returns a persistent read-only view of the tree in constant time. The implementation of the data structure uses shared mutable data between the tree and its snapshots and applies *copy-on-write* to lazily unshare subtrees. The sharing is not observable by a client, which is reflected in our specification. I verified the full functional correctness of the snapshotable tree data structure in Chapter 10 implemented in Java using separation logic embedded into Coq.

In Chapter 11 I present a Java implementation of a solution to the point location problem using snapshotable trees, and develop a specification and outline a proof of the correctness of the query method.

I use access permissions and typestate, a lightweight verification approach, to verify the correct API usage of the snapshotable tree data structure in Chapter 12.

1.5 Background

The foundation for this dissertation is mathematics, more precisely the field of formal logic. We use model theory, one branch of formal logic, to describe the semantics of a programming language. We use proof theory, another branch of formal logic, to reason about the correctness of programs.

In this section I describe the object-oriented programming language used in this dissertation, its operational semantics, and its Hoare rules for verification. Afterwards, I present how we handle method calls and where separation logic gets into the game. I show the foundations of proof assistants and how we formalised the presented programming language in Coq. Finally, I discuss software engineering with a focus on software maintenance.

1.5.1 Programming Language

I use a small object-oriented programming language throughout this dissertation. My programming language, which I refer to as *SimpleJava*, is a subset of the widely used programming language Java. The focus of SimpleJava is object-orientation in a sequential setting.

In my opinion object-orientation is a paradigm which inherits from procedural programming, an extension to imperative programming. An imperative program consists of a series of statements that *modify* program *state*. Procedural programming organizes a series of statements into a named block, also called procedure or method. A procedure is an abstraction, which enables modular development of software. In the object-oriented paradigm a developer combines both data fields and procedures into objects. One family of object-oriented languages are class-based languages. Both Java and SimpleJava are class-based. A program is a combination of classes, which contain data fields and procedures. A class contains compile time information about a program, whereas an object is a run time instance of a class.

The unique characteristics of object-oriented programming are not generally defined, instead I reproduce here its key features as described by Pierce [98, Chapter 18]. In addition to classes there are *interfaces*, which do not contain program code, but describe only operations and signatures thereof. An interface may have multiple implementations with different representations. A method invocation involves looking up the method name at run time in a method table of the object on which the method was invoked. This process is called *dynamic dispatch*. Object-oriented programs use *encapsulation* to hide the internal representation of an object from the outside world (other objects). This encapsulation greatly improves readability and maintainability, because the internal representation can only be accessed and modified within a small, localizable region of the program. Instead of requiring a concrete class as method argument, an interface can be used. With *subtyping* a class implementing an interface can be passed to a method where the implemented interface is expected. Most representations behave in a similar way, this is called behavioural subtyping [77]. Classes can share parts of their program code from their superclass by *inheritance*. A method body can invoke another method of the same object by using *open recursion*: a reference to the object is implicitly passed around by using the **this** parameter (also called *late binding*).

I claim that SimpleJava is an object-oriented programming language. In SimpleJava both *interfaces* and classes can be defined. The method call rule implement the *dynamic dispatch* discipline. A class in SimpleJava is a list of fields and methods, thus it can be used to *encapsulate* the internal representation. Whilst SimpleJava supports *subtyping*, there is no support for *inheritance* (or subclassing). An object method in SimpleJava receives an implicit **this** parameter, which is *late bound*.

In type theory the programming language Featherweight Java [57] is very popular. SimpleJava has slightly different features than Featherweight Java: SimpleJava supports interfaces, and control flow operations such as assignment, conditional and loop. In contrast, Featherweight Java does not include interfaces and also no assignment. Featherweight Java includes inheritance by subclassing. It includes a cast operation, that claims a certain run time type for an object and possibly fails during execution. The syntax of SimpleJava is close to Featherweight Java in the areas where features are common. Because Featherweight Java does

$$
\begin{aligned}
\mathcal{P} &::= \overline{\mathcal{I}}\ \overline{\mathcal{C}} \\
\mathcal{I} &::= \textbf{interface}\ \texttt{interface-name}\ \textbf{extends}\ \overline{\texttt{interface-name}}\ \{\overline{\mathcal{N}}\} \\
\mathcal{C} &::= \textbf{class}\ \texttt{class-name}\ \textbf{implements}\ \overline{\texttt{interface-name}}\ \{\overline{\texttt{T}}\ \overline{\texttt{fname}}, \overline{\mathcal{M}}\} \\
\\
\mathcal{N} &::= \texttt{K}\ \texttt{T}\ \texttt{method-name}\ (\overline{\texttt{T}}\ \overline{a}) \\
\mathcal{M} &::= \texttt{K}\ \texttt{T}\ \texttt{method-name}\ (\overline{\texttt{T}}\ \overline{a})\{s; \textbf{return}\ r\} \\
\\
s &::= s_1; s_2 \mid x := y.f \mid x.f := e \mid x := e \mid y := x.m(\overline{e}) \mid \textbf{skip} \mid \textbf{assert}(e) \\
&\quad \mid x := \textbf{new}\ C() \mid \textbf{while}\ e\ \textbf{do}\ \{s\} \mid \textbf{if}\ e\ \textbf{then}\ \{s_1\}\ \textbf{else}\ \{s_2\} \\
e &::= e_1\ aop\ e_2 \mid e_1\ bop\ e_2 \mid !e \mid x \mid \textbf{true} \mid \textbf{false} \mid \textbf{this} \mid \textbf{null} \mid number \\
aop &::= + \mid * \mid - \\
bop &::= \&\& \mid || \\
\\
\texttt{T} &::= \textbf{boolean} \mid \textbf{int} \mid \textbf{void} \mid C \\
\texttt{K} &::= \textbf{static} \mid \epsilon
\end{aligned}
$$

Figure 1.1: The syntax of SimpleJava. Keywords are bold. The metavariables x and y range over program variables, f over field names, m over method names, C over class names, e_1, e_2 and r over expressions, s_1 and s_2 over statements. A program $\mathcal{P}$ is a list of classes $\mathcal{C}$ and interfaces $\mathcal{I}$. The metavariables `interface-name`, `class-name`, `fname`, `method-name` and a are identifiers, which have to conform to the usual Java rules.

not include assignment, and thus does not need a store, the operational semantics of SimpleJava are different from Featherweight Java.

The popular object-oriented programming language used in the wild is Java. Java includes a lot of features which are not supported in SimpleJava, such as inheritance, non-local exits, arrays, parametric polymorphism, exceptions, methods with variable arity, and more. SimpleJava is a strict subset of Java, which enables us to use Java development tools and Java compilers for SimpleJava.

The syntax of SimpleJava is shown in Figure 1.1, which defines a program $\mathcal{P}$ to be a list of interfaces $\mathcal{I}$ and classes $\mathcal{C}$. An interface consists of a name, a list of superinterfaces and a list of method signatures $\mathcal{N}$. A class consists of a name, a list of implemented interfaces, and a list of

typed fields `T fname` and method implementations $\mathcal{M}$. A method signature $\mathcal{N}$ consists of optional modifiers, the type of its return value, the method name, and a list of arguments `T` a. A method implementation $\mathcal{M}$ consists of optional modifiers, the type of its return value, its method name, a list of arguments `T` a, and the body, which is a list of statements s and a single return statement. There are several kinds of statements s: sequential composition, field read, field write, assignment, method call, skip, assert, allocation, while loop, and conditional. Expressions e do not access the heap, and are either binary arithmetic operations *aop*, binary boolean operations *bop*, negation of an expression, variable access, or a literal constant: true, false, this, null, or a literal number. Binary arithmetic operations *aop* are addition, multiplication, and subtraction. The binary boolean operations *bop* are the logical "and" and the logical "or". A type `T` is either a boolean, an integer, a trivial uninhabited type **void**, or a class C. The only supported modifier `K` is **static** to indicate a class method, rather than an instance method. A class method has no access to fields, cannot call instance methods, and does not receive an implicit **this** as argument.

In contrast to Java, SimpleJava does distinguish between expressions and statements. At certain places, such as arguments of a method call or guards of conditionals and loops, SimpleJava only supports expressions. This restriction can be eluded by a semantics-preserving program transformation which introduces temporary variables. In fact, Kopitiam implements this transformation.

1.5.2 Operational Semantics

Operational semantics describe how a program is executed. I focus on big-step operational semantics, which describe how the overall results of a program is obtained. For each statement a rule is defined which describes the reduction of the statement. I describe the big-step operational semantics of SimpleJava's statements.

I define, similar to Featherweight Java, the field and method lookup auxiliary functions in Figure 1.2. In contrast to the corresponding Featherweight Java definitions, the SimpleJava auxiliary functions do not recurse, because all fields and methods are defined in the current class definition, since SimpleJava does not support subclassing. The function *fields* returns the list of field names of a specific class. I use the function

$$\frac{\textbf{class } C \textbf{ implements } \overline{\texttt{interface-name}}\ \{\overline{\texttt{T}}\ \overline{\texttt{fname}}, \overline{\mathcal{M}}\}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{fname}}}$$

$$\frac{\begin{array}{c}\textbf{class } C \textbf{ implements } \overline{\texttt{interface-name}}\ \{\overline{\texttt{T}}\ \overline{\texttt{fname}}, \overline{\mathcal{M}}\} \\ \texttt{T}_\texttt{r}\ \texttt{m}(\overline{\texttt{T}_\texttt{a}}\ \overline{\texttt{a}})\{\texttt{s}; \textbf{return } r\} \in \overline{\mathcal{M}}\end{array}}{\mathit{mBody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{a}},\ \texttt{s}; \textbf{return } r)}$$

Figure 1.2: Auxiliary functions for field and method body lookup.

mBody to lookup the pair of argument names and the method body. A global class table in which defined classes and interfaces are stored is implicitly assumed.

We use run time type information for dynamic dispatch. SimpleJava has no static typing rules, because they are very similar to Featherweight Java and our Coq formalisation uses SimpleJava without static types.

The imperative part of my call-by-value big step operational semantics for SimpleJava is shown in Figure 1.3. The rules only describe the successful reduction of statements; I do not describe erroneous reductions. The reduction $\mathcal{H}, \mathcal{S}, s \longrightarrow \mathcal{H}', \mathcal{S}'$ takes a triple consisting of a global store, the heap $\mathcal{H}$, a local store, the stack $\mathcal{S}$, and a statement s, and reduces to a pair consisting of a modified heap $\mathcal{H}'$ and a modified stack $\mathcal{S}'$.

The first rule R-Seq handles sequencing of statements: if a statement s_1 reduces to the modified heap $\mathcal{H}'$ and the modified stack $\mathcal{S}'$, and another statement s_2 reduces this heap and stack to a modified heap $\mathcal{H}''$ and a modified stack $\mathcal{S}''$, then the sequence of s_1 and s_2, written $s_1; s_2$, reduces from the initial heap $\mathcal{H}$ and initial stack $\mathcal{S}$ to the final heap $\mathcal{H}''$ and the final stack $\mathcal{S}''$.

The heap $\mathcal{H}$ represents the global store, and is a partial function from a pair of address and field name to value:

$$\mathcal{H} : (\mathit{address}, \texttt{fname}) \rightharpoonup_{\mathit{fin}} \mathit{val}$$

I use ι as metavariable to range over addresses. The stack $\mathcal{S}$ represents local storage, and is a partial function from a variable name to its value:

$$\mathcal{S} : \mathit{var} \rightharpoonup_{\mathit{fin}} \mathit{val}$$

$$\frac{\mathcal{H}, \mathcal{S}, s_1 \longrightarrow \mathcal{H}', \mathcal{S}' \qquad \mathcal{H}', \mathcal{S}', s_2 \longrightarrow \mathcal{H}'', \mathcal{S}''}{\mathcal{H}, \mathcal{S}, s_1; s_2 \longrightarrow \mathcal{H}'', \mathcal{S}''} \text{ R-Seq}$$

$$\frac{[\![e]\!]_{\mathcal{S}} = v}{\mathcal{H}, \mathcal{S}, x := e \longrightarrow \mathcal{H}, \mathcal{S}[x \mapsto v]} \text{ R-Assign} \qquad \frac{}{\mathcal{H}, \mathcal{S}, \textbf{skip} \longrightarrow \mathcal{H}, \mathcal{S}} \text{ R-Skip}$$

$$\frac{[\![e]\!]_{\mathcal{S}} = \textbf{true}}{\mathcal{H}, \mathcal{S}, \textbf{assert}(e) \longrightarrow \mathcal{H}, \mathcal{S}} \text{ R-Assert}$$

$$\frac{\mathcal{H}, \mathcal{S}, s \longrightarrow \mathcal{H}', \mathcal{S}' \qquad \begin{array}{c}[\![e]\!]_{\mathcal{S}} = \textbf{true} \\ \mathcal{H}', \mathcal{S}', \textbf{while } e \textbf{ do } s \longrightarrow \mathcal{H}'', \mathcal{S}''\end{array}}{\mathcal{H}, \mathcal{S}, \textbf{while } e \textbf{ do } s \longrightarrow \mathcal{H}'', \mathcal{S}''} \text{ R-While}$$

$$\frac{[\![e]\!]_{\mathcal{S}} = \textbf{false}}{\mathcal{H}, \mathcal{S}, \textbf{while } e \textbf{ do } s \longrightarrow \mathcal{H}, \mathcal{S}} \text{ R-WhileEnd}$$

$$\frac{[\![e]\!]_{\mathcal{S}} = \textbf{true} \qquad \mathcal{H}, \mathcal{S}, s_1 \longrightarrow \mathcal{H}', \mathcal{S}'}{\mathcal{H}, \mathcal{S}, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \longrightarrow \mathcal{H}', \mathcal{S}'} \text{ R-IfTrue}$$

$$\frac{[\![e]\!]_{\mathcal{S}} = \textbf{false} \qquad \mathcal{H}, \mathcal{S}, s_2 \longrightarrow \mathcal{H}', \mathcal{S}'}{\mathcal{H}, \mathcal{S}, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \longrightarrow \mathcal{H}', \mathcal{S}'} \text{ R-IfFalse}$$

Figure 1.3: Imperative part of call-by-value big step operational semantics of SimpleJava.

I use $[\![e]\!]_{\mathcal{S}}$ to notate the evaluation of the expression e using the stack $\mathcal{S}$. Such an evaluation on the stack is pure, and computes a value. It is trivial and not shown here.

An assignment R-Assign of an expression e to a stack variable x is evaluated by first evaluating the expression e on the stack $\mathcal{S}$ to its value v. The original stack $\mathcal{S}$ is extended so that the value of the variable x is v. A **skip** statement has no effect, as shown by the R-Skip rule. To reduce an assertion R-Assert, the asserted expression e has to evaluate to **true** on the stack $\mathcal{S}$. If it does not, no rule applies and the program does not execute further. I use two rules for a loop: R-While is invoked if the guard e evaluates on the stack to **true**. The loop body s is reduced once, followed by a reduction of the loop with the modified heap $\mathcal{H}'$ and

$$\frac{\mathcal{S}(y) = \iota \qquad \mathcal{H}(\iota, f) = v}{\mathcal{H}, \mathcal{S}, x := y.f \longrightarrow \mathcal{H}, \mathcal{S}[x \mapsto v]} \text{ R-FieldRead}$$

$$\frac{\mathcal{S}(x) = \iota \qquad (\iota, f) \in dom(\mathcal{H}) \qquad [\![e]\!]_{\mathcal{S}} = v}{\mathcal{H}, \mathcal{S}, x.f := e \longrightarrow \mathcal{H}[(\iota, f) \mapsto v], \mathcal{S}} \text{ R-FieldWrite}$$

$$\frac{\begin{array}{c} mBody(m, C) = (\overline{a},\ s; \textbf{return}\ r) \\ \mathcal{H}, [\overline{a \mapsto [\![e]\!]_{\mathcal{S}}}], s \longrightarrow \mathcal{H}', \mathcal{S}' \qquad [\![r]\!]_{\mathcal{S}'} = v \end{array}}{\mathcal{H}, \mathcal{S}, y := C.m(\overline{e}) \longrightarrow \mathcal{H}', \mathcal{S}[y \mapsto v]} \text{ R-StaticCall}$$

$$\frac{\begin{array}{c} \mathcal{S}(x) = \iota \qquad \iota :: C \qquad mBody(m, C) = (\overline{a},\ s; \textbf{return}\ r) \\ \mathcal{H}, [\textbf{this} \mapsto \iota, \overline{a \mapsto [\![e]\!]_{\mathcal{S}}}], s \longrightarrow \mathcal{H}', \mathcal{S}' \qquad [\![r]\!]_{\mathcal{S}'} = v \end{array}}{\mathcal{H}, \mathcal{S}, y := x.m(\overline{e}) \longrightarrow \mathcal{H}', \mathcal{S}[y \mapsto v]} \text{ R-DynCall}$$

$$\frac{\iota \text{ fresh} \qquad (\iota, \_) \notin dom(\mathcal{H}) \qquad fields(C) = \overline{f_i}}{\mathcal{H}, \mathcal{S}, x := \textbf{new}\ C() \longrightarrow \mathcal{H}[\overline{(\iota, f_i) \mapsto \textbf{null}}], \mathcal{S}[x \mapsto \iota]} \text{ R-New}$$

Figure 1.4: Object-oriented part of call-by-value big step operational semantics of SimpleJava.

modified stack $\mathcal{S}'$. If the guard e evaluates to **false**, in rule R-WhileEnd, then the reduction of the loop terminates. A conditional statement either reduces to the consequent s_1 in rule R-IfTrue if the guard e evaluates to **true** or to the alternative s_2 in rule R-IfFalse if the guard e evaluates to **false**.

In Figure 1.4 I show the operational semantics of the object-oriented part of SimpleJava. The dynamic type operation, $\iota :: C$, used in rule R-DynCall, asserts that reference ι refers to an object of class C. The rule R-New handles object allocation by assigning the value **null** to all fields of the instantiated class. This is safe because SimpleJava is untyped, thus the values **null**, 0 and **false** are freely convertible.

1.5.3 Hoare Logic

Hoare logic [55] is a methodology to reason about imperative programs, which consist of a series of statements. The main idea is to have a set of

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}}\ \textsc{Skip} \qquad \frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1; s_2\ \{R\}}\ \textsc{Seq}$$

$$\frac{\{P \wedge e\}\ s\ \{P\}}{\{P\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{P \wedge \neg e\}}\ \textsc{While}$$

$$\frac{\{P \wedge e\}\ s_1\ \{Q\} \qquad \{P \wedge \neg e\}\ s_2\ \{Q\}}{\{P\}\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \{Q\}}\ \textsc{If} \qquad \frac{P \vdash e}{\{P\}\ \textbf{assert}(e)\ \{P\}}\ \textsc{Assert}$$

$$\frac{}{\{P\}\ x := e\ \{\exists v. P[v/x] \wedge x = e[v/x]\}}\ \textsc{Assign}$$

Figure 1.5: Imperative part of Hoare rules for SimpleJava.

assertions, which are logic predicates, and applying for each statement in the series its corresponding Hoare rule. I use an intuitionistic logic in this dissertation, which means that an assertion is considered true only if a proof of it can be constructed. The conclusion of a Hoare rule consists of a Hoare triple $\{P\}\ s\ \{Q\}$: a precondition $\{P\}$, the statement s for which this rule is applicable, and a postcondition $\{Q\}$. The precondition $\{P\}$ describes the assertions which must be valid before the statement s can be executed. After execution of the statement the assertions of the postcondition $\{Q\}$ holds. I define Hoare rules for partial correctness: only if a statement terminates, its Hoare triple has to be valid. My Hoare rules for SimpleJava are derived from the operational semantics presented in the last section. Furthermore, the rules are formulated in a way that simplifies formalised reasoning about them. An example is that the amount of side conditions is minimized, which can be seen in the FieldRead rule later.

In Figure 1.5 I define the Hoare rules for the imperative part of SimpleJava. The first rule is Skip, which is read as: assuming nothing, if the assertion P holds, then after **skip** is executed P still holds. The second rule Seq is used for sequencing statements: if the rule for a statement s_1 gives rise to Q being valid, assuming that P was valid before execution, and another statement s_2 requires the same Q being valid as precondition and ensures that R is valid as postcondition, then the execution

of $s_1; s_2$ in sequence requires the validity of the precondition P and ensures the validity of the postcondition R. For a loop we define the rule While: the loop body s is only executed if the guard e was true, thus its precondition requires $P \wedge e$ to be valid. The postcondition of the loop body ensures that P is valid, because we have no information about the guard. This gives rise to the triple: the precondition P has to be valid, and after execution of the loop the postcondition $P \wedge \neg e$ is valid. If the guard e would have been valid, the loop would have executed once more. The rule for a conditional If with a guard e under the assumption that the precondition of the consequent s_1 requires $P \wedge e$ to be valid and ensures that the postcondition Q is valid, and the precondition of the alternative s_2 requires $P \wedge \neg e$ to be valid and its postcondition ensures that Q is valid, the precondition of the conditional requires P to be valid and the postcondition ensures Q to be valid. An assertion Assert has a precondition that requires P gives rise to e being valid, and ensures the validity of the same postcondition P. I do not consider the case that the assertion is not valid, because I am only considering partial correctness. The last rule Assign is used for assignment: the precondition requires P to be valid, the postcondition ensures the validity of the assigned value v, which is substituted for every x in P and an assertion that $x = e[v/x]$, which substitutes all x for v in the expression e.

An important rule in Hoare logic is rule of consequence, which allows to strengthen the precondition and to weaken the postcondition of a Hoare triple.

$$\frac{P \vdash P' \qquad \{P'\}\ s\ \{Q'\} \qquad Q' \vdash Q}{\{P\}\ s\ \{Q\}}\ \textsc{Consequence}$$

There are two challenges we need to address next: the first is method calls and the second is shared mutable data. To reason about method calls, we extend the syntax of SimpleJava to include pre- and postconditions for method definitions. I use separation logic, an extension of Hoare logic, to reason about object-oriented features of SimpleJava which mutate the heap like instantiation, field read, and field write.

1.5.4 Method Calls

I extend the syntax for method definitions to include a pre- and postcondition, as shown in Figure 1.6. Both the pre- and the postcondition

$$\begin{aligned}\mathcal{M} ::= \ & \textbf{requires} : spec_{pre} \\ & \textbf{ensures} : r.spec_{post} \\ & \texttt{K T method-name } (\overline{\texttt{T}}\ \overline{\texttt{a}})\{s; \textbf{return } r\}\end{aligned}$$

Figure 1.6: Extended SimpleJava syntax extended with method precondition $spec_{pre}$ and postcondition $spec_{post}$, which are logical assertions.

$$\frac{\textbf{class } C \textbf{ implements } \overline{\texttt{interface-name}}\ \{\overline{\texttt{T}}\ \overline{\texttt{fname}}, \overline{\mathcal{M}}\} \quad \textbf{requires} : P,\ \textbf{ensures} : r.Q,\ \texttt{T}_\texttt{r}\ \texttt{m}(\overline{\texttt{T}_\texttt{a}}\ \overline{\texttt{a}})\{\texttt{s}; \textbf{return } r\} \in \overline{\mathcal{M}}}{mSpec(\texttt{m}, C) = \{P\}\ \_\ \{r.Q\}}$$

Figure 1.7: The auxiliary function $mSpec$.

$$\frac{\triangleright\ mSpec(C, m) = \{P\}\ \_\ \{r.Q\} \qquad mBody(C, m) = (\overline{p}, s)}{\{P[\overline{e}/\overline{p}]\}\ y := C.m(\overline{e})\ \{\exists v.Q[y, \overline{e}[v/y]/r, \overline{p}]\}}\ \textsc{StaticCall}$$

$$\frac{\triangleright\ mSpec(C, m) = \{P\}\ \_\ \{r.Q\} \qquad mBody(C, m) = (\overline{p}, s)}{\{x :: C \wedge P[x, \overline{e}/\overline{p}]\}\ y := x.m(\overline{e})\ \{\exists v.Q[y, x[v/y], \overline{e}[v/y]/r, \overline{p}]\}}\ \textsc{DynCall}$$

Figure 1.8: Hoare rules for static and dynamic method calls.

are logical assertions. The postcondition $r.spec_{post}$ is a unary predicate, which receives the actual return value r of the method. For a trivial return value (of type **void**) we omit it, instead we use solely $spec_{post}$.

The function $mSpec$ is defined in Figure 1.7, which returns the Hoare triple of a given method $\texttt{m}$ and a given class C. The return value r of the method is in scope of the postcondition Q.

I use the auxiliary function $mSpec$ to define the Hoare rules for dynamic and static method calls in Figure 1.8. The rule StaticCall for a static method call substitutes the actual parameters $\overline{e}$ for the formal parameters $\overline{p}$ in the precondition P by $[\overline{e}/\overline{p}]$. In the postcondition Q the binding y to the returned value is substituted with the concrete returned

value, in addition to the actual parameters for formal parameters substitution. The substitution in the rule DynCall for a dynamic method call is a bit more complicated, because it has to deal with the implicit **this** argument passed around.

Methods in most object-oriented programming languages may call each other or themselves recursively, due to late binding this behaves well and is also well defined. From a verification perspective we have a problem: we cannot simply postulate the correctness of a method without verifying it, and we also cannot linearize the order of the methods in which to verify their correctness, due to circularity and self-recursion. I use the *later* operator ($\triangleright$) from Gödel-Löb logic to support recursion. The intuition behind it is that assuming a formula P is valid in a future world, the fact that it is true can already be used now. This is a common trick to handle recursion [4].

$$\frac{\triangleright\, P \vdash P}{\vdash P}\ \textsc{Löb}$$

1.5.5 Separation Logic

Separation logic [100] was discovered by Reynolds and O'Hearn. It is an extension to Hoare logic that integrates the concept of a heap. A heap is modelled as a partial function from a pair of an address and field name to a value. I use the points-to relation $\iota.f \mapsto v$ to express the assertion that field f of object ι points to value v in the heap.

The separating conjunction connective $P * Q$ expresses the validity of both P and Q, each for a disjoint heap. It gives rise to the frame rule:

$$\frac{\{P\}\ s\ \{Q\} \qquad \forall x \in fv\ R.\ s\ \textit{does not modify}\ x}{\{P * R\}\ s\ \{Q * R\}}\ \textsc{Frame}$$

The frame rule allows local reasoning: if a triple $\{P\}\ s\ \{Q\}$ is valid for small heaps P and Q, it is also valid for bigger heaps $P * R$ and $Q * R$, as long as the extension R is disjoint from the small heaps and s does not modify any free variable of R. Thus, the frame rule enables modular reasoning. A proof for a class can be reused in another program if the class is encapsulated in such a way that no other parts of the program can directly access and modify the heap used by the class.

$$\frac{P \vdash y.f \mapsto e}{\{P\}\ x := y.f\ \{\exists v.P[v/x] \wedge x = e[v/x]\}}\ \textsc{FieldRead}$$

$$\frac{}{\{x.f \mapsto \_\}\ x.f := e\ \{x.f \mapsto e\}}\ \textsc{FieldWrite}$$

$$\frac{}{\{\mathit{true}\}\ x := \mathbf{new}\ C()\ \{\forall^* f \in \mathit{fields}(C).\ x.f \mapsto \mathbf{null}\}}\ \textsc{New}$$

Figure 1.9: Hoare rules that account for the use of a heap.

In Figure 1.9 I define the Hoare rules for the remaining statements of SimpleJava: field read, field write and allocation. The rule FieldRead requires that the object y with field f points to some value e, which is then used as the new value of x by substitution in the postcondition. Writing to a field (FieldWrite) requires that the object x with field f points to something (_). This something is replaced by the assigned expression e in the postcondition. The rule for allocation New uses the iterating separating conjunction ($\forall^*$) to express that all fields of the freshly allocated object x point to **null**.

To deal with abstraction and interfaces some extensions were proposed by Parkinson et al. [94, 95]. To reason about data abstraction via quantification over resource invariants I use a *higher-order* separation logic, as proposed by Biering et al. [20]. A higher-order separation logic allows for quantification over predicates in both the assertion logic (the logic of pre- and post-conditions) and the specification logic (the logic of Hoare triples).

In this dissertation I use an intuitionistic version of separation logic. In intuitionistic separation logic the truth of assertions is closed under heap extension, which is appropriate for a garbage-collected language such as Java, rather than a language with manual memory management, such as C.

I show the inference rules of intuitionistic separation logic in Figure 1.10: $*$ is commutative and associative, the relationship between the

$$\frac{}{P * Q \vdash Q * P} \qquad \frac{}{(P * Q) * R \vdash P * (Q * R)} \qquad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

$$\frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R} \qquad \frac{P \vdash Q}{P * R \vdash Q * R} \qquad \frac{}{P * emp \vdash P} \qquad \frac{}{P \vdash P * emp}$$

Figure 1.10: Inference rules for separation logic

separating implication $\mathrel{-\!\!*}$ and $*$, that a fresh heap R can be added, and that the empty heap *emp* can be added or removed.

1.5.6 Proof Assistants

The correspondence discovered by Curry and Howard [56] is that proofs are programs. More precisely, constructive logic can be seen as the typed lambda calculus. A theorem can be seen as a type, and evidence of its correctness as data. There is a one-to-one correspondence between introduction and elimination rules, and cut elimination corresponds to normalization. Furthermore, implications are functions, a sum type represents a disjunction, a product type a conjunction, a universal quantification corresponds to a dependent function (Π type), an existential quantification can be seen as the dependent pair (Σ type).

This correspondence is used by proof assistants: verifying a proof is equivalent to type checking a program, using an expressive enough type system. Coq, the proof assistant that we use in this dissertation, is an LCF-style proof assistant. This means that only a small kernel has to be trusted, while further theories, like set theory and natural numbers, are built on top of the kernel.

To be convinced that a proof is correct, there are only a few components which have to be trusted: the kernel of the proof assistant, the programming language compiler which compiled the proof assistant to assembly, the operating system executing the proof assistant and the hardware on which the operating system is executed. None of these components are specific to the concrete proof. All of them come off-the-shelf, fulfill a more general purpose, and can be tested extensively on other tasks. In contrast to a proof written on paper, which has to be

reviewed by mathematicians, the just-mentioned components are tested on a daily basis by a huge number of people around the world.

1.5.7 Charge!

The presented operational semantics for SimpleJava and higher-order separation logic is formalised in the Charge! framework [11, 12] in the proof assistant Coq. Charge! is roughly 10000 lines of Coq code, which was developed by Jesper Bengtson, Jonas Jensen and Filip Sieczkowski. Charge! and Kopitiam were developed side by side within the research project "Tools and Methods for Scalable Software Verification" (Grant 09-065888 from the Danish Research Council for Technology and Production (DFF-FTP)). While Charge! enables interactive reasoning about SimpleJava programs, Kopitiam serves as the frontend for a developer. An interesting aspect of Charge! in contrast to other formalisation projects is that Charge! allows to reason about programs written in SimpleJava, rather than verifying only metatheoretical properties of SimpleJava. While Charge! supports abstract predicates to reason abstractly about interfaces, this is not supported by Kopitiam yet. An interface in Charge! is represented as a Coq definition in the specification logic.

1.5.8 Software Engineering

More than 30 years ago, Lehman [73] described that most software development time is spent in maintenance. Software documentation is important for its maintenance. Knuth developed literate programming [65] to integrate development of programs with documentation. Several contemporary programming languages and integrated development environments supporting their development promote documentation facilities embedded into the source code as comments, such as Javadoc. The rise of such integrated documentation systems ensure us a lot of software with up to date documentation.

Verification of the correctness of software is also important, furthermore it suffers from the same problem: to enable well maintained proofs we need integration of verification and software development into the same tool. In Chapter 8 I motivate a tight integration and present a solution by adapting the programming workflow.

1.6 Disclaimer

This dissertation contains several research papers previously published. The chapters for which this is the case mention venue and collaborators just below the title of the chapter. Only minor modifications, such as spelling fixes, adjustments to the layout, and unification of the citations were applied. Dated footnotes in this dissertation constitute updates to the published papers.

Chapter 2

Design Space of Verification Tools

There is a large variety of verification tools, which differ in certain aspects. While there are tools available which apply lightweight verification technologies, such as checking for the absence of null pointer dereferences, checking for the absence of out of array bounds accesses, or checking for the compliance to a specified API protocol, we focus on full functional verification tools. Furthermore, we only take tools which statically verify the correctness into consideration. We do not study verification tools which emit run time checks to achieve correctness proofs.

We conducted a feature analysis, a method for discovering and representing commonalities amongst related software [61], of verification tools for imperative programming languages. By conducting this feature analysis we discovered five important aspects of verification tools: the user interface, the verification mechanisms they use, the specification logic, the supported programming language, and the trusted code base. Figure 2.1 shows a feature diagram [103] based on our analysis. In the following we describe the different features in more detail and present verification tools with these features.

2.1 User Interface

We distinguish two fundamentally different approaches towards verification of programs: fully automated or interactive. A verification tool with full automation receives the program to be verified and a specification thereof as input, and applies techniques to verify the specifications.

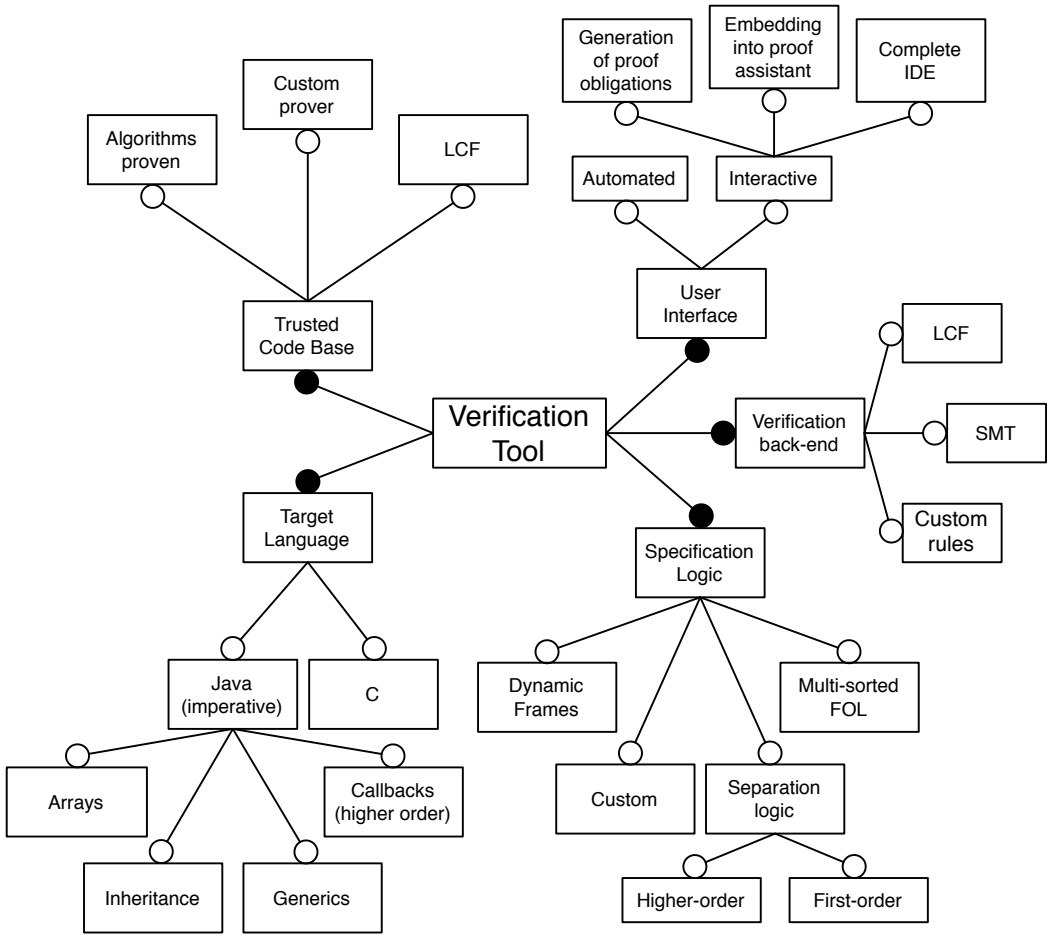


Figure 2.1: Feature diagram of verification tools. The most important aspects of features are indicated by a filled circle.

These techniques include symbolic execution, proof search, entailment checking, and inference of anti-frames. Examples of fully automated tools are Spec# for C# [7] and ESC/Java2 for Java [36]. Some automated tools also use separation logic, like Smallfoot [13], SLAyer [14], Space Invader [32], Infer [30], and jStar [45].

We distinguish between three types of interactive verification: either (1) proof obligations are generated, which have to be proven in a separate proof assistant, or (2) the verification tool is tightly integrated into a proof assistant, or (3) the verification tool provides a complete integrated development environment.

The first approach has the disadvantage that when software evolves, the proofs have to be done again. This approach is used by Jahob [70] and Loop [113]. Loop translates JML-annotated Java programs to proof obligations for the interactive proof assistant PVS. Jahob uses both interactive and automated theorem provers to discharge its proof obligations. It uses the interactive proof assistants Coq and Isabelle, and the automated theorem provers SPASS, CVC3 and Z3.

The second approach, integration into a proof assistant, has the disadvantage that a custom programming language is integrated into a proof assistant. First the software for which correctness is verified has to be developed in that custom programming language and furthermore the equivalence of the original software with the translated software has to be shown. This approach is used by Ynot [35] and Bedrock [34]. Both extend the proof assistant Coq with an imperative programming language.

The third approach, a complete IDE, has been used for designing development environments such as KeY [1], VeriFast [58], Why3 [23], Mobius PVE [9] and Dafny [74]. These verification tools differ in the features of the IDE not related to verification, as compilers, debuggers, unit testers, profilers, etc. Also, these verification tools differ in other aspects of our feature model.

2.2 Verification Back-end

We distinguish verification tools by the back-end they use. The back-end can be a LCF-style interactive proof assistant, an SMT solver, or a solver which can be extended by a developer with custom proof rules. The powerfulness of verification back-ends is different, while using an interactive proof assistant a developer can use a higher-order logic, SMT solvers can only discharge first-order logic assertions. Also, other features of our feature analysis are affected by the verification back-end: an interactive proof assistant requires an interactive user interface, and the trusted code base of a solver extended with custom proof rules is enlarged by those custom proof rules, unless they are proven sound.

Several verification tools use a LCF-style interactive proof assistant: the verification tools which integrate into a proof assistant (Ynot [35] and Bedrock [34]) and also several other as Jahob [70], Why3 [23], JACK [8].

The satisfiability modulo theories (SMT) problem is a decision problem for logical formulae, which use theories expressed in (classical) first-order logic with equality. Fast automatic solvers for SMT are available, like Z3. The advantage is that the solving is completely automated, no manual interaction is required. Tools like VeriFast [58] and Dafny [74] use a SMT solver to discharge proof obligations.

The verification tool jStar [45] lets a user extend the proof rules which are used. This implies the problem that these user-defined rules can be unsound, since there are no checks for their soundness.

2.3 Specification Logic

A wide variety of logics, which vary in expressiveness, are used to write specifications. We distinguish between custom logics, which might not have formalised semantics, first-order logics, which can only express first-order predicates, dynamic frames, and separation logic. Parkinson and Summers [93] recently described the relationship between first-order separation logic and implicit dynamic frames, which can be translated into each other. Both separation logic and dynamic frames model the concept of storage, the heap. While some tools use a first-order separation logic, other tools use a higher-order separation logic.

The tool Jahob [70] uses a multi-sorted first-order logic to express specifications.

First-order separation logic is used in verification tools such as VeriFast [58], jStar [45], Infer [30], SLAyer [14], Smallfoot [13], and Space Invader [32]. The advantage of first-order separation logic is that decision procedures can be used for reasoning, whereas the disadvantage is that higher-order programs cannot be specified easily.

Higher-order separation logic is more expressive than first-order separation logic, but automated reasoning about higher-order logic is undecidable. Thus, a verification tool using higher-order logic needs manual interaction to discharge proof obligations. One example for a tool using higher-order separation logic is Ynot [35].

Dynamic Frames [62] are a mechanism to express pre- and post-condition with reasoning about the heap. The verification tools VeriCool [106] and Dafny [74] use dynamic frames.

Some tools use custom logics, like ESC/Java2 for Java [36], which uses JML and code contracts [7, 48]. While these are powerful, there is no formalised meaning of these custom logics such as JML.

2.4 Target Language

We distinguish between two sorts of programming languages: either low-level imperative programming languages that allow pointer arithmetics, such as C, or object-oriented programming languages like Java. Both programming language families have different challenges: while in C support for reasoning about pointer arithmetics is crucial, modularity by interfaces is important in Java.

The verification tool VeriFast [58] targets both C and Java programs. Infer [30] is automated and targets C programs.

A huge number of verification tools target an imperative object-oriented Java-like programming language, with fewer or more advanced features like arrays, inheritance, generics, and callbacks. Jahob [70], VeriCool [106] and jStar [45] target Java programs. Dafny [74] targets a custom imperative programming language with arrays and generics.

2.5 Trusted Code Base

The trusted code base describes how much software needs to be trusted to consider a program to be correct. If the verification tool itself does not have any formalisation, its complete implementation is part of the trusted code base of proofs accomplished with the tool. Some verification tools are implemented inside of LCF-style proof assistants, others have the correctness of their core algorithms verified inside of a LCF-style proof assistant.

Smallfoot [13] is an example of a verification tool where the correctness of its core algorithms are proven [107] with a proof assistant. The core algorithms of VeriFast have also been proven correct [115] in Coq.

Other verification tools rely on a custom developed prover, whose correctness is shown either on paper or has to be trusted. Examples of this kind are VeriCool [106] and Dafny [74].

Some verification tools emit proofs which can be checked by a LCF-style proof assistant. The trusted code base of these tools is very small,

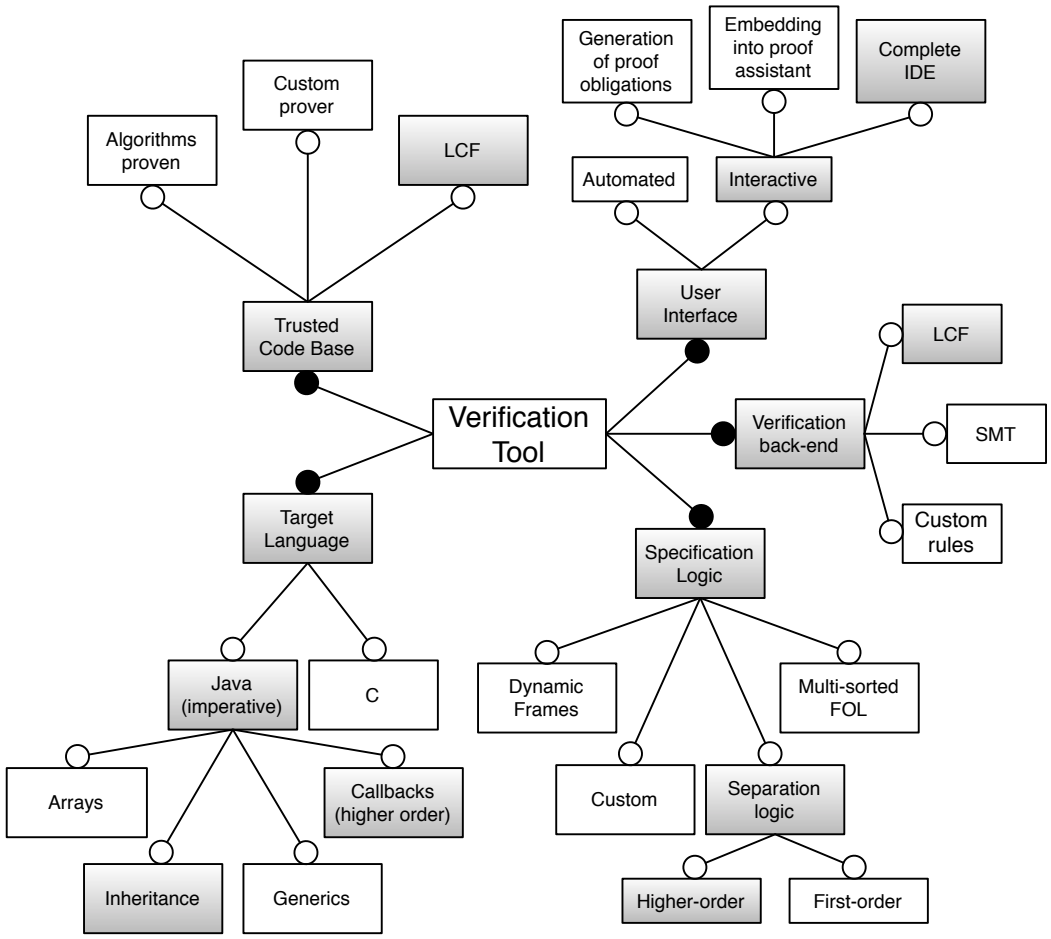


Figure 2.2: Feature diagram of verification tools. The highlighted features implemented in Kopitiam.

since only the core of the LCF-style proof assistant has to be trusted. Without the verification tool itself, but only by using the proof assistant, the validity of proofs of correctness of software can be checked. Ynot [35] and Bedrock [34] are examples of this kind of tools, among others.

2.6 Which Features Does Kopitiam Implement?

The Kopitiam tool presented in this dissertation fits as follows in Figure 2.2 into the landscape of verification tools. The different aspects are: it is an interactive tool which is integrated into the industry-grade

IDE Eclipse. In the back-end the proof assistant Coq is used to discharge proof obligations interactively. Using Coq has several advantages: the trusted code base is small, the theories developed in Coq can be used, the modularization and abstraction features of Coq are immediately available in Kopitiam. This shows some unique characteristics of Kopitiam; other verification tools implement a custom prover which sometimes needs to be extended with abstraction features, and its code base still has to be trusted. Kopitiam uses Charge! [11, 12], a higher-order separation logic embedding into Coq. Charge! implements nested triples [104], which allows to embed Hoare triples into the assertions and gives rise to reason about first-class functions. So far we have not used Charge! and Kopitiam to reason about callbacks and delegates, but that should be straightforward using Svendsen et al.'s work [110]. The target language of Kopitiam is Java with inheritance, so far neither arrays nor generics are supported, which . Kopitiam's trusted code base is the LCF-style proof assistant Coq, because Kopitiam emits proofs which can be checked using only the proof assistant Coq with Charge!. Charge! contains an operational semantics for Java, which is part of the trusted code base. Additionally, Kopitiam's translation of Java programs into Coq definitions has to be trusted. The target group of Kopitiam are developers which have a background in formal methods.

Chapter 3

Related Work

In Chapter 2 we presented a feature analysis of verification tools, which should be considered to be part of the related work of this dissertation.

Shape Analysis Shape analysis attempts to discover the shapes of data structures in the heap. This analysis works reasonably well for simple data structures which are known upfront, like linked lists or binary search trees. There is also work on using separation logic with shape analysis [44, 31].

A very interesting approach using shape analysis, separation logic and real-world data structures [72] reasons about the correctness of a data structure found in the Linux scheduler. This data structure combines a tree to insert tasks with a priority queue to schedule tasks by using several references in a task object.

Data structure fusion [54] is a slightly different approach: instead of a data structure with shared mutable state only a basic implementation is specified in a relational algebra. A fuse operation, working on relational indexes, specifies where the physical data structure has sharing. The advantage of this approach is that a basic data structure implementation is easier to verify correct, and the fuse operation is correct by construction.

Permissions Boyland [26] developed fractional permissions, which are used by a variety of research groups to track access to objects. The idea is to provide exclusive access to an object using 1, and shared access with fractions thereof. With exclusive access both read and write operations

are safe, whereas a permission with a fraction smaller than 1, but greater than 0, can safely access the object read only. A fraction of 0 gives no access to the object. Annotations, which are either keywords (“shared”, “unique”, “immutable”) or actual fractions, allow developer to annotate references with their access.

Access permission moves the burden to the developer and allow lots of automated tools to do automatic verification, as long as the data structures are straightforward enough. In a higher-order logic fractional permissions can be embedded, thus fractional permissions are less powerful than higher-order logics.

In a different line of research [24] I integrated access permissions into the type system of a functional language (SML and F#) in order to make the program automatically concurrency safe. The main idea is that if there is unique access to a cell, this can be done concurrently without leading to race conditions.

Another area of research, so called object propositions [91], use access permissions and linear logic to verify the correctness of programs in an automated way. I extended object propositions to verify the composite pattern. During the verification I discovered some shortcomings of an earlier verification of the composite pattern [19] using typestate.

Dynamic Frames Dynamic frames were initially developed by Kassios [62] to reason about imperative programs. More recently, Parkinson [93] compares separation logic and implicit dynamic frames. Dynamic frames and a first-order separation logic with fractional permissions are isomorphic, if considering a total heap in the separation logic. In separation logic, reasoning is usually done on a partial heap. Several verification tools use implicit dynamic frames to reason about object-oriented software.

Immutability If a reference is declared as immutable, similarly to **final** in Java, this information allows compiler optimizations. Also, reasoning about immutable data structures is easier. An interesting approach is taken by using immutability for verification [39], which allows immutability annotations of predicates, and verifies those first. In a second phase, the mutatable predicates are verified.

Chapter 4

Future Work

I cannot predict the future. Nevertheless, the research papers of this dissertation contain ideas for future work, which are briefly summarized in this chapter.

The Eclipse plugin Kopitiam, which is presented in this dissertation, requires some improvements regarding usability. Once it is more useful and developers of Coq theories can use it for incremental compilation, navigation, dependency management, etc., it will be released and announced to a wider public. The Danish research council already accepted a grant proposal (FTP, number 12-132607, grant holder is Jesper Bengtson) to fund a research assistant for two years to develop Kopitiam further, in combination with funding a postdoc for three years to improve Charge!, our Coq formalisation. The preliminary evaluation of Kopitiam's usefulness should be extended by further evaluating the use of Kopitiam by teaching future courses using Kopitiam. In Section 8.7 I present several ideas for future work for Kopitiam.

Once our separation logic embedding Charge! is capable of handling callbacks and generics, the verification of the solution to the point location can be formalised. This will show the practicality of our paper-based verification approach and if successful, it will result in a proof with a high confidence. Another challenge is to develop a cost model for our Java semantics and then, using this, to verify the time and space complexity of the solution to the point location problem.

In this dissertation I have looked into sequential data structures which contain non-observable sharing. The snapshotable tree data structure in Chapter 10 is a case study for such a sequential data structure

with non-observable sharing. There are more advanced implementations of the same data structure available, which we did not yet prove. To achieve correctness proofs of non-observable sharing, I looked into approaches to verification of concurrent algorithms. One extension for future work is to enrich our Java semantics with concurrency and adapt the verification approaches that I used successfully for sequential programs.

Verification of the correctness of object-oriented programs is still a challenging research area. I do not believe that someone can grab object-oriented programs off-the-shelf and enhance them with proofs of their correctness. Rather, these programs have first to be refactored, so that they have a clearer structure and contain less mutable state. Evidence for this is described in Chapter 11. An alternative approach to reasoning about imperative programs is to reimplement them in another programming language with dependent types, like Agda or Idris. These programming language have a powerful type system which can be used to carry proofs of programs.

A highly related topic to this dissertation is integrated development environments for dependently typed programming languages. Coq is a dependently typed programming language, but the types are mostly used for proofs, rather than programming. There is much more static information available from dependently typed programs, due to more precise types. In the future I aim to look into using this information to write excellent integrated development environments for dependently typed programming languages, such as Idris. These environments should not only support the development of a program, but also the maintainence of a program over its entire lifetime.

Chapter 5

Conclusion

I have implemented several generations of Kopitiam, which is a plugin for the industry-grade integrated development environment Eclipse that integrates the well-known proof assistant Coq to support verification of the correctness of software. I also conducted a qualitative evaluation about the usefulness of one recent generation with positive results. Although there are still issues to address, I am confident that Kopitiam is a robust foundation for future verification tools.

Furthermore, I conducted case studies on full functional verification of data structures which share parts of the heap in a non-trivial way. The correctness of one implementation of snapshotable trees is fully formalised in Coq using Charge!. I laid the basis for verifying the correctness of a solution to the point location problem. By working towards full functional verification of the point location problem I learned to *smash the state*, e.g. for the comparator of line segments, before writing a specification and proof for a program. The favorable programming style if one aims to verify the correctness of her program is to use mutable state only when necessary. Programs have to be either designed and implemented with verification in mind, or refactored with this aim.

I confirm the main thesis of this dissertation: modular incremental interactive verification of correctness of object-oriented software is achievable and can be tightly integrated into the software development workflow. Kopitiam is a research prototype which provides evidence for this thesis. The case studies I conducted with Charge! provide evidence that it is mature enough to verify the correctness of data structures with non-trivial shared state.

Part II

Research Papers: Kopitiam

Chapter 6

Kopitiam: Modular Incremental Interactive Full Functional Static Verification of Java Code

Originally published in: NASA Formal Methods 2011 [79]

Abstract

We are developing Kopitiam, a tool to interactively prove full functional correctness of Java programs using separation logic by interacting with the interactive theorem prover Coq. Kopitiam is an Eclipse plugin, enabling seamless integration into the workflow of a developer. Kopitiam enables a user to develop proofs side-by-side with Java programs in Eclipse.

6.1 Introduction

It is challenging to reason about object-oriented programs, because these contain implicit side effects, shared mutable data and aliasing. Reasoning with Hoare logic always has to consider the complete heap, which does not preserve the abstractions of the programming language. Separation logic [100] extends Hoare logic to allow modular local reasoning about programs with shared mutable state.

Coq [15] is an interactive theorem prover based on the calculus of constructions with inductive definitions. Kopitiam generates proof obli-

gations from specifications written in Java, which the user needs to discharge by providing Coq proof scripts. A proof script is a sequence of tactics.

The contribution is Kopitiam, a tool combining the following verification properties:

- **Modular** Extensions of a verified Java library can rely on the specification of the library, without reverifying the library.
- **Incremental** While parts of the code can be verified and proven, other parts might remain unverified, and development of proofs and code can be interleaved, as in Code Contracts [48].
- **Interactive** Automated proof systems like jStar [45] are limited in what they can prove. We use an interactive approach where the user discharges the proof obligations using provided tactics, thus Kopitiam does not limit what a user can prove.
- **Full functional** Given a complete, precise formal specification the proof shows that the implementation adheres to its specification.
- **Static** The complete verification is done at compile time, without execution of the program. Other code verification approaches, like design by contract [83], may depend on run time checks. Especially in mission critical systems, compile time verification is indispensable, since a failing run time check would be disastrous.

The structure of the paper is: we give an overview of Kopitiam in Section 6.2, demonstrate a detailed example in Section 6.3, relate Kopitiam to similar tools in Section 6.4, and in Section 6.5 conclude and present future work.

6.2 Overview of Kopitiam

Kopitiam provides an environment that is familiar to both Java programmers and Coq users. Coq developers use Proof General (based on Emacs) or CoqIDE (a self-hosted user interface). Many Java programmers use an IDE for development, the major Java IDEs are Eclipse and IntelliJ. To integrate seamlessly into the normal development workflow we develop Kopitiam as a plugin for Eclipse, so a developer does

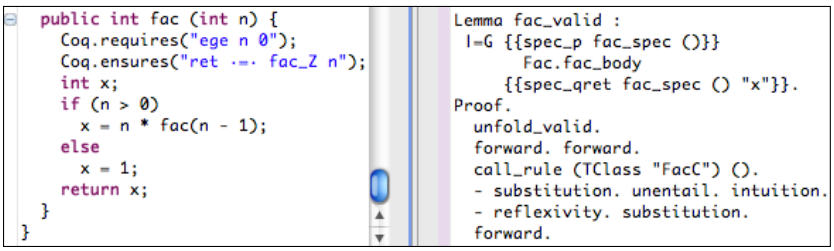


Figure 6.1: Java and Coq editor side-by-side; closeup of Coq editor in Figure 6.2

not have to switch tools to prove her code correct. We base Kopitiam on Eclipse because it is open source, popular and easily extendible via plugins. While an Eclipse integration for Coq [33] already exists, Kopitiam provides a stronger integration of Java code and Coq proofs. This is achieved by a single intermediate representation for both code and proofs. A change to either code or proof directly changes this intermediate representation.

In Figure 6.1 Kopitiam is shown. It consists of a standard Eclipse Java editor on the left and a specially developed Coq proof editor on the right. The content of the Java editor is the method `fac`, a recursive implementation of the factorial function. The Java code contains a call to `Coq.requires` and a call to `Coq.ensures`, whose arguments are the pre- and postcondition of the method. The right side shows the Coq lemma `fac_valid`, stating that factorial fulfills its specification, together with parts of the proof script (full code in Section 6.3). Due to the single intermediate language, Kopitiam reflects every change to the content of one editor to the other editor, e.g. a change to the specification on the Java side changes the Coq proof obligation.

Kopitiam consists of a Java parser, with semantic analysis, a transformer to SimpleJava (presented in Section 6.2.2), a Coq parser, and communication to Coq via standard input and output. All these parts are expressible in a functional way, so we chose Scala [92] as the implementation language of Kopitiam. Scala is a type-safe functional object-oriented language supporting pattern matching. It compiles to Java bytecode, allowing for seamless integration with Eclipse (every Scala object is a Java object and vice versa). Kopitiam is open source under the Simplified BSD License and available at `https://github.com/hannesm/Kopitiam`.

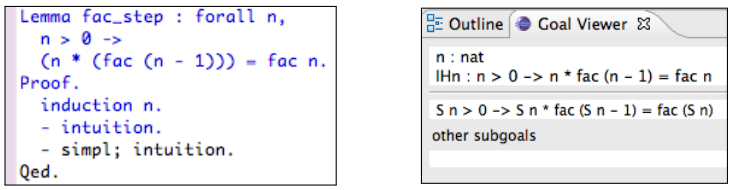


Figure 6.2: Coq editor and goal viewer of Kopitiam, closeup of Figure 6.1

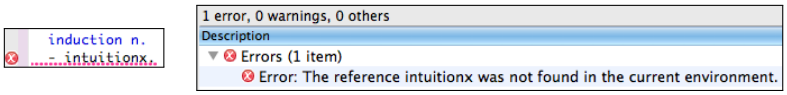


Figure 6.3: Coq proof script containing an error and Eclipse's problems tab

6.2.1 Coq Editor and Goal Viewer

To develop proofs, Kopitiam provides a Coq editor and a goal viewer, shown in Figure 6.2. The Coq code on the left side states the lemma `fac_step`: for all `n`, `n` greater than 0 implies that `n * fac(n - 1)` equals `fac(n)` (lines 1-3). All except the last 2 lines of the Coq code that have been processed by Coq (highlighted in blue in Kopitiam, the unprocessed ones are black). The goal viewer on the right side shows the current state of proof assumptions, proof obligations and subgoals. The current state is after doing induction over `n` and discharging the base case using the `intuition` tactic. The remaining proof obligation is the induction step.

As in other Coq user interfaces, there are buttons (not shown) to step forward and backward through the proof.

If Coq signals an error while processing, this error is highlighted in Kopitiam. Figure 6.3 shows on the left side the erroneous Coq proof script next to Eclipse's corresponding problems tab. Errors are indicated by red wiggly lines, similar to the way programming errors are displayed in Eclipse.

6.2.2 The SimpleJava Programming Language

We formalized SimpleJava, a subset of Java, and implemented it using a shallow embedding in Coq [12]. SimpleJava syntax is a prefix (S-expression) notation of Java's abstract syntax tree. Dynamic method dispatch is the core ingredient of object oriented programming, and sup-

```
class FacC {
  int fac (int n) {
    Coq.requires("ege n 0");
    Coq.ensures("ret ·=· facZ n");
    int x;
    if (n > 0)
      x = n * fac(n - 1);
    else
      x = 1;
    return x;
  }
}
```

Figure 6.4: Java code implementing factorial, using the single class `FacC` containing the single instance method `fac`. The method calls to `Coq.requires` and `Coq.ensures` form the specification. The arguments to these method calls are transformed into Coq definitions, shown in Figure 6.7.

ported by SimpleJava. A SimpleJava program consists of classes and interfaces. An interface contains a set of method signatures and a set of interfaces, that it inherits from; a class consists of a set of implemented interfaces, a set of fields, and a set of method implementations. A method body consists of a sequence of statements (allocation, conditional, loop, call, field read, field write and assignment) followed by a single return statement. Automatic transformation of unstructured returns to a single return would impose method-global control flow changes; and the SimpleJava code would distract the Java programmer while proving.

6.3 Example Verification of Factorial

An example program is the factorial, shown in Figure 6.4. Figure 6.5 shows the SimpleJava code, automatically translated by Kopitiam. A call (lines 3-5) consists of the return value binding (`x`), the receiver (`this`), the method (`fac`), the argument list and the receiver class (`TClass "FacC"`).

In Figure 6.6 the fixpoint `fac` is defined, which is the common factorial function on natural numbers. Our Java code uses integers, so we additionally need `facZ`, which extends the domain of `fac` to integers.

The specification of a program consists of specifications for all classes and interfaces. An example specification of method `fac` is shown in

```
(cif (egt (var_expr "n") 0)
  (cseq
    (ccall "x" "this" "fac"
      ((eminus (var_expr "n") 1))
      (TClass "FacC"))
    (cassign "x"
      (etimes
        (var_expr "n")
        (var_expr "x"))))
  (cassign "x" 1))
```

Figure 6.5: SimpleJava code implementing factorial, translated by Kopitiam from the Java code in Figure 6.4.

```
Fixpoint fac n :=
  match n with
    | S n => (S n) * fac n
    | 0 => 1
  end.

Definition facZ :=
  fun (n:Z) =>
    match ((n ?= 0)%Z) with
      | Lt => 0
      | _ => Z_of_nat(fac(Zabs_nat n))
    end.
```

Figure 6.6: Factorial implemented in Gallina, Coq's pure functional programming language.

Figure 6.7, whose code is automatically generated by Kopitiam from the Java code (Figure 6.4). The precondition (line 3 of both Figures) requires that the parameter `n` must be equal or greater (`ege`) than 0. The postcondition (line 4 of both Figures) ensures that the returned value (`ret`) is equal to `facZ n`. The bottom block of Figure 6.7 defines `Spec`, which connects the specification `fac_s` to the actual program, class `FacC`, method `fac`.

Figure 6.8 shows the hand-written proof that the Java implementation of factorial satisfies its specification. The proof uses the `forward` tactic [3]. This extracts the first Hoare triple; the resulting proof obligation

```
Definition fac_s :=
  Build_spec unit (fun _ =>
   (ege "n" 0,
   ((("ret":expr) ·=· (facZ ("n":expr))):asn))).

Definition Spec := TM.add
  (TClass "FacC")
  (SM.add "fac" ("n" :: nil, fac_s) (SM.empty _))
  (TM.empty _).
```

Figure 6.7: Specification of the Java factorial in Coq, translated by Kopitiam from the calls to the static `Coq.requires` and `Coq.ensures` in Figure 6.4.

```
Lemma fac_valid : |=G {{spec_p Fac_spec.fac_spec ()}}
 Fac.fac_body {{spec_qret Fac_spec.fac_spec () "x"}}.
Proof.
  unfold_valid. forward. forward.
  call_rule (TClass "FacC") ().
  - substitution. unentail. intuition.
  - reflexivity. substitution.
  forward. unentail. intuition. subst. simpl.
    rewrite Fac_spec.facZ_step; [reflexivity | omega].
  forward. unentail. intuition. subst.
    destruct (Z_dec (val_to_int k) 0).
    assert False; [|intuition]. destruct s; intuition.
    rewrite e. intuition.
  Existential 1:=().
Qed.
```

Figure 6.8: Coq proof script verifying the correctness of our Java factorial in Figure 6.4 regarding our Coq implementation in Figure 6.6.

(Hoare triple) is the original precondition combined with the extracted postcondition, the remaining statement sequence, and the original postcondition. If the extracted precondition cannot be discharged trivially, the user has to do it. After applying `forward` twice (line 4, for `cif` and `cseq`), the proof obligation for the call is discharged by the `call_rule` tactic (line 5).

Name	T	Language	Specification logic	Automation
Krakatoa	sta	Java; While	multi-sorted FOL	several provers
Ynot	sta	HO imp	separation logic	Coq tactics
jStar	sta	Java; Jimple	separation logic	proof rules, SMT
Spec#	dyn	C#	C#/Java	run time assertions
Dafny	inc	imp + generics	Boogie	Z3 (SMT-solver)
Kopitiam	inc	Java; SimpleJava	separation logic	Coq tactics

Table 6.1: Comparison of verification tools

6.4 Related Work

Several currently available proof tools are compared in Table 6.1. Only Krakatoa [49], jStar [45] and Kopitiam target Java. Krakatoa uses Why, which uses a simple While language where mutable variables cannot be aliased. The automated proof system jStar targets Jimple [112], a Java intermediate language built from Java bytecode. Kopitiam directly translates from a subset of Java source code to SimpleJava.

Different code contracts [83] implementations focus on C# (Code Contracts [48]) and Java (JML [29]). Code contract implementations translate some non-trivial specifications to run time checks, while we focus on static verification. The integration of code contracts in an IDE is beneficial, as the developer can incrementally develop code and proofs in the same environment. An example for an industrial grade IDE with code contracts is the KeY tool [1], based on UML and OCL. Code Contracts [48] do not focus on full functional correctness, while some JML tools such as Mobius [9] do. In contrast to those tools, we use separation logic, thus a user does not need to specify frame conditions.

Dafny [74] is a proof tool for an imperative programming language supporting generics and algebraic data types, but not subtyping. Dafny is well integrated into Microsoft Visual Studio and also allows incremental proofs. It provides a multi-sorted first-order logic as specification logic.

Ynot [35] uses a shallow embedding in Coq for a higher-order imperative programming language without inheritance. Thus to verify code with Ynot the program has to be reimplemented in the Ynot tool.

The jStar [45] tool is fully automated and does a proof search on available proof rules, which are extensible by the user. A user can introduce unsound proof rules, since these are treated as axioms and are

not verified. Moreover it is difficult to guide the proof search in jStar, since the order of rules matters. Both Ynot and Kopitiam use the proof assistant Coq, in which proof rules have to be proven before usage.

6.5 Conclusion and Future Work

We are developing Kopitiam, an Eclipse plugin for interactive full functional static verification of Java code using separation logic. Our implementation is complete enough to prove correctness of factorial and in-place reversal of linked lists. We currently do not handle the complete Java language, e.g. unstructured returns and `switch` statements. Class to class inheritance is also not supported. Kopitiam does not support more advanced Java features like generics and exceptions.

We plan to integrate more automation: We will provide context aware suggestions, a technique widely used in Eclipse for code completion, for specifications, whose syntax we also plan to improve. We will provide separation logic lemmas and tactics for Coq, allowing the user to focus on the non-trivial proof obligations. We also want the user to discharge separation logic proof obligations instead of exposing the Coq layer.

We are also working on more and larger case studies ranging from simple object-oriented code (`Cell` and `ReCell` from [95]), to the composite pattern and other verification challenges [116], to real-world data structures like Linked Lists with Views [60] and Snapshotable Trees [46], to the C5 collection library [66], the extensive case study of our research project.

Acknowledgement

We want to thank Peter Sestoft, Jesper Bengtson, Joe Kiniry and the anonymous reviewers for their valuable feedback.

Chapter 7

Kopitiam – a Unified IDE for Developing Formally Verified Java Programs

Originally published in: ITU Technical Report 167, May 2013 [81]

Joint work with: Jesper Bengtson; ITU Copenhagen

Abstract

We present Kopitiam, an Eclipse plugin for certifying full functional correctness of Java programs using higher-order separation logic. Kopitiam extends the Eclipse Java IDE with an interactive environment for program verification, powered by the general-purpose proof assistant Coq. Moreover, Kopitiam includes a development environment for Coq theories, where users can define program models, and prove theorems required for the program verification.

7.1 Introduction

It is difficult to design software that is guaranteed to work. For most software, correctness is inferred by extensive and costly testing, but this can only prove the presence of errors, not their absence. For safety critical systems this is unsatisfactory. The ideal is to specify the desired behaviour of a program using logics and mathematics, and then formally prove that the program satisfies its specification; this guarantees that all cases have been covered and that no stone has been left unturned. Formal methods have had a renaissance in recent years, and they are being

incorporated into tools that are used in different parts of the software development cycle, but they are typically highly specialised and focus on ensuring that a program satisfies a key property such as deadlock freedom, memory safety, or termination.

In this paper we present Kopitiam, an extension to the Java development environment of Eclipse that allows programmers to prove full functional correctness of Java programs. Kopitiam is designed according to the following set of design philosophies:

1. Users should be able to develop programs, their models, their specifications and their proofs of correctness incrementally and side by side.

2. The program logic must be expressive enough to reason about features such as mutable state, shared data structures and pointer aliasing.

3. The specification language must be expressive enough to describe the behaviour of all programs that we want to certify

4. All proofs that users write must be checked automatically.

5. No bug in the tool itself may break soundness – a buggy tool may fail to certify a program, but it must never claim that a defective program is correct.

6. The integrated development environment (IDE) must include the tools that software developers are accustomed to, such as compilers, profilers, unit testers, and debuggers.

Software development is an incremental process, and it is important that programs are kept synchronised with their proofs of correctness. This is ensured by point 1. The danger is otherwise that a proof proves properties about an outdated version of the program.

Point 2 requires that we support some dialect of separation logic [100]. Since its conception, separation logic has been used to great success for modular reasoning about programs written in languages using shared data, pointers, aliasing, and destructive updates, including object-oriented languages [20, 94, 95, 114]. By *modular reasoning* we mean

that the specification of a program can constrain itself to the state that the program actually acts upon, and not the entire proof state.

Point 3 requires that the proofs are developed interactively as the heuristics required to find them will generally be undecidable. In practice, this requires that the user has access to an interactive proof assistant such as Coq [111], Isabelle [90], or HOL [52]. These theorem provers support a higher-order logic that is expressive enough to reason not only about specification languages but also about the semantics of programming languages. Combined, this allows them to certify theorems that prove that programs satisfy their specifications with respect to the semantics of the programming language. Another distinct advantage of the interactive proof assistants is that they have a kernel that automatically certifies the proofs produced by the user, as required by point 4. Moreover, this kernel is very small and is the most complicated piece of code that actually has to be trusted (apart from a few minor parsing and output routines). This provides the reliability required by point 5. The kernel will reject a proof that is incorrect, even if it was produced by the theorem prover or by another part of the tool, due to a bug.

Finally, it is important that the developer has access to a state of the art development environment for the actual software development, as required by point 6. Even though our ultimate goal is to prove the absence of bugs in our programs, the process of formal verification is currently prohibitive and testing and debugging are very useful tools to get a program in such a shape that it can reasonably be assumed to be bug free before undertaking the verification effort.

Kopitiam

Kopitiam is an Eclipse plugin that integrates the Java development environment of Eclipse with the general-purpose interactive proof assistant Coq. The software verification process is split into two distinct parts; proofs about the program specification, which are done in a Coq perspective, and proofs about the actual program, which are interleaved with the program itself in the Java IDE. Program verification is interactive. The user steps through the program, in the style of a debugger, proving that all prerequisites are met for every statement. The workflow is described in Figure 7.1.

As a back-end, Kopitiam uses Charge! [11, 12], a framework for verifying Java programs using an intuitionistic higher-order separation logic in Coq. Charge! provides tactics that automatically discharge commonly occurring proof obligations, allowing the user to focus on the interesting rather than the tedious aspects of program verification.

Contributions

Our main contribution is that we provide one uniform framework for formal verification of Java programs that integrates an industry-grade IDE with a general-purpose interactive proof assistant. More specifically, Kopitiam includes:

- *An extension to the Eclipse Java development environment* that allows users to annotate their code with specifications for each method (§7.2.2, 7.3.2). The user steps through the program one statement at a time, executing each symbolically and observing how the proof state changes. All steps are verified by Charge!. If necessary, these steps are interleaved with Coq code to prove that the current proof state satisfies the conditions required to make the next step.
- *A development environment for Coq theories* (§7.2.2, 7.3.1). This perspective allows users to state theorems in Coq, and construct proofs of their correctness interactively, much like Proof General, or CoqIDE. This mode is used to define the model of the program, and prove all necessary properties needed for verification.
- *Support for passing Java program variables as arguments to predicates defined in the Coq standard library* (§7.2.2 p. 59). This allows models and specifications to use predicates and terms that have not been designed with software verification in mind.
- *Support for proof certificates*. Once a program has been verified with Kopitiam, we produce a proof certificate (§7.2.2 p. 61), which contains the program and a theorem and corresponding proof that the program is correct. This proof certificate is checkable by Coq using Charge!.

To the best of our knowledge, no other tool integrates an industry-grade IDE like Eclipse with an interactive proof assistant this closely.

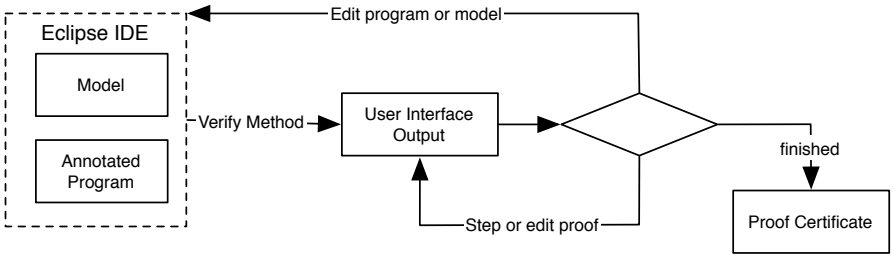


Figure 7.1: Kopitiam workflow. The user writes an annotated program in the Java perspective, and a model of the program in the Coq perspective. Each method is verified one at a time. The user steps through the statements of the method, using inline Coq commands to update the proof state where necessary. Kopitiam automatically produces a certificate of correctness when all methods have been verified.

There are other tools that target software verification, but those that use interactive proof assistants either verify the programs completely in a proof assistant or generate theory files from annotated code (and thus lose the ability to keep code and proofs synchronised). Another approach is to build the entire development and verification environment from scratch, which invariably leads to fewer features than those available when building on already well-established IDEs and proof assistants. An extensive comparison with related work is presented in Section 7.5.

Kopitiam is released under the BSD license. Examples and installation instructions are available at `http://itu.dk/research/tomeso/kopitiam/`.

7.2 Using Kopitiam

We demonstrate program verification in Kopitiam by certifying a small library for linked lists with two classes: an inner class `Node` for the list elements and the class `List` for the lists themselves. The public API of the `List` class consists of a `length` method, which defers its work to the auxiliary recursive `nodeLength` method in the `Node` class, an `add` method that adds an element to the head of the list, and finally a method `reverse` for in place list reversal. The source code is presented in Figure 7.2.

To verify this library, we need a model for each class, and a specification for each method. The models, and the proofs about them, are

```
class List {
  static class Node {
    int value;
    Node next;

    public int nodeLength() {
      int r = 1;
      Node n = next;
      if (n != null) {
        r = n.nodeLength();
        r = r + 1;
      }
      return r;
    }
  }

  Node head;

  public int length() {
    Node h = head; int r = 0;
    if (h != null) r = h.nodeLength();
    return r;
  }

  public void add (int n) {
    Node x = new Node(); x.value = n;
    Node h = head; x.next = h;
    head = x;
  }

  public void reverse () {
    Node old = null; Node lst = head;
    while (lst != null) {
      Node tmp = lst.next; lst.next = old;
      old = lst; lst = tmp;
    }
    head = old;
  }
}
```

Figure 7.2: A small list library. The library has one inner Node class, which is used for each list element. The Node class allows us to differentiate between an empty list (where head is null) and the null pointer.

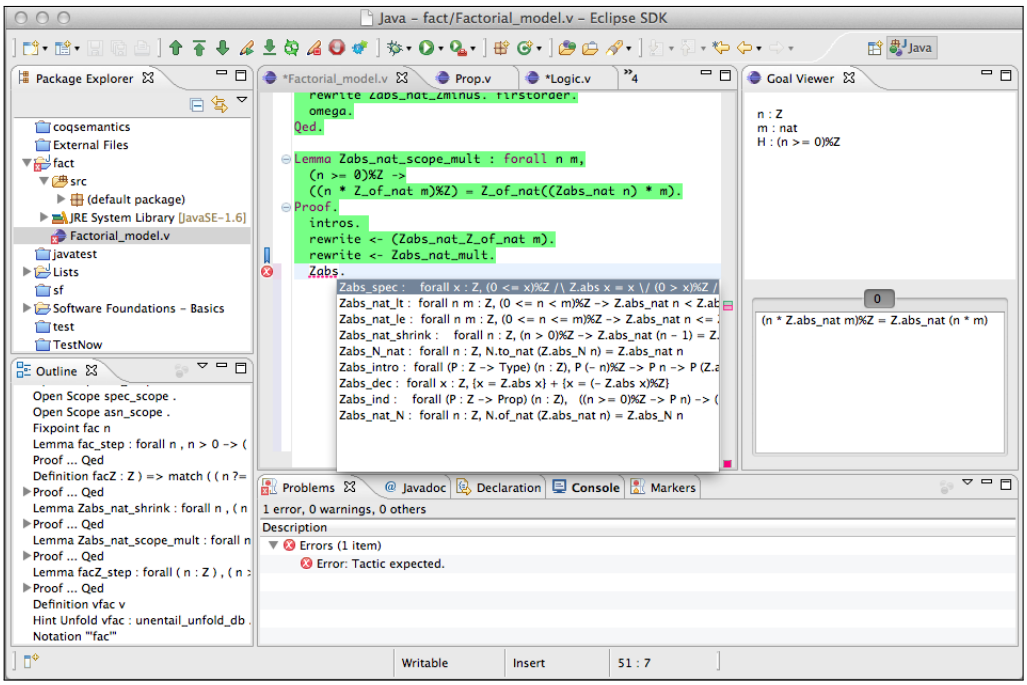


Figure 7.3: Screenshot of the Coq perspective in Kopitiam. To the left, the package explorer and the Coq theory outline; in the middle, the theory file and the error window; to the right, the goal viewer which has the Coq context at the top, and the current goal at the bottom. The 0 indicates that there is currently only one subgoal. In the theory file, the green background indicates how far the theory has been processed by Coq. Moreover, the term Zabs has been underlined and marked as an error since no lemma or tactic with that name exists; by pressing Ctrl-Space we obtain a list of possible completions.

developed in the Coq perspective. The specifications, and the proofs of the actual source code, are developed in the Java perspective. We discuss each in turn.

7.2.1 Coq Perspective

The Coq perspective is a development environment for Coq theories in Eclipse. It highlights the command currently being processed, as well as the commands that have been processed by Coq, with a yellow and a green background respectively. If the proof script contains an error, the faulty code is highlighted with a red squiggly line, and the error is presented in the error window. A goal viewer shows the current proof obligation and further subgoals. Kopitiam also supports keyboard

shortcuts for common tasks, such as stepping through the proofs, and syntax highlighting. We also support features not commonly found in theorem prover interfaces such as folding of proofs, an outline for Coq theories, and code completion that suggests commands and lemmas, including the ones developed by the user. The perspective is powerful enough to develop standard Coq theories, and is not limited to program verification. A screenshot is presented in Figure 7.3.

Defining the Program Model

The theoretical foundation of Kopitiam is the Charge! framework, which verifies Java-like programs in Coq using a higher-order intuitionistic separation logic. Separation logic allows users to reason about the shape and contents of the heap. For this exposition, we only require two construct specific to separation logic: the points-to predicate ($\mapsto$) and the separating conjunction ($*$). The predicate $c.f \mapsto v$ states that an object instance referenced by c has a field f that currently contains the value v; the predicate $p * q$ states that the predicate p holds for one part of the heap and the predicate q holds for another disjoint part of the heap. The $*$-operator gives rise to the characteristic frame rule of separation logic

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}\ c \textit{ does not mention } r$$

which states that a command c that runs in the state p and terminates in q, can run in an environment where a disjoint r is present as long as no command in c modifies r. In general, the Hoare-triple $\{p\}\ c\ \{q\}$ states that if the command c starts from state p and terminates, it will satisfy the state q.

The semantics of Java in Charge! is untyped. A single type `val` in Coq represents Java types. Separation logic predicates have the type `asn`. There is also a `typeof` predicate, where `typeof p C` states that the dynamic type of the object reference `p` is the class `C`.

For our list library, we define one model for the inner `Node` class, and one for the `List` class. Throughout the paper, we use standard mathematical notation in Coq code, and not the ASCII code used by Charge!.

```
Fixpoint Node_list (p : val) (lst : list val) : asn :=
  match lst with
```

```
    | nil       ⇒ p = null
    | x :: xs ⇒ typeof p Node ∧ p.value↦x ∗
                  ∃v : val. p.next↦v ∗ Node_list v xs
  end.
```

Lists are modelled using the lists from the Coq standard library. The predicate Node_list p lst is defined recursively over lst and creates one instance of Node on the heap for every element in lst, where every instance points to the next respective element in the list.

The predicate List_rep p lst states that p has the dynamic type List and that the head field of p points to an object modelled by the Node_list predicate.

```
Definition List_rep (p : val) (lst : list val) : asn :=
    typeof p List ∧ ∃h : val. p.head↦h ∗ Node_list h lst
```

Proof Development

The Coq perspective is also used to develop the meta-theoretical properties that we require for the program model. For this list library, we need a lemma that states that the only list that can model the null-pointer is the empty list.

```
Lemma lst_ptr_null: ∀k lst. k = null ⟶ (Node_list k lst ⊢ lst = [])
Proof.
   ...
Qed.
```

This lemma states that if k is null and k is modelled by a list *lst*, then *lst* must be the empty list. This lemma is proven in Coq by case analysis on *lst*.

7.2.2 Java Perspective

The Java perspective of Kopitiam is an extension of the Java editor. We have extended the syntax with antiquotation brackets <% and %>, between which Kopitiam-specific code is placed. These brackets and their

```
<% lvars: xs : list val %>
<% requires: `List_rep "this"/V `xs %>
<% ensures: `List_rep "this"/V (`cons "n"/V `xs) %>
public void add (int n) { ... }

<% lvars: xs : list val %>
<% requires: `List_rep "this"/V `xs %>
<% ensures: `List_rep "this"/V `(rev xs) %>
public void reverse () { ... }

<% lvars: v : val, xs : list val %>
<% requires: "this"/V.`next ↦ `v ∗ `Node_rep `v `xs
<% ensures: "r", "this"/V.`next ↦ `v ∗ `Node_rep `v `xs ∧
                "r"/V == `((length xs) + 1)%>
public int nodeLength () { ... }

<% lvars: xs : list val %>
<% requires: `List_rep "this"/V `xs %>
<% ensures: "r", `List_rep "this"/V `xs ∧
                "r"/V == `(length xs) %>
public int length () { ... }
```

Figure 7.4: Specification of the methods in the list library. Note how they all use the backtick operator (`), or the /V notation, so that predicates that normally take values as arguments take program variables instead. This applies not only to our own List_rep and Node_rep predicates, but also to functions from the Coq standard library like cons. The ensures clauses in nodeLength and length both have the return variable "r", which indicates that any value returned will be assigned to this variable.

contents are ignored by the rest of Eclipse and all of its standard functionality is maintained. We use the Java perspective to write programs and their specifications, and ultimately prove that the programs satisfy their specifications.

Specifications

Specifications of methods have the form

```
<% lvars: x1 : T1, ..., xn : Tn %>
<% requires: P x1 ... xn %>
<% ensures: Q x1 ... xn %>
```

where x_1 to x_n are logical variables universally quantified over P and Q, T_1 to T_n are their types, P is the precondition and Q is the postcondition of the method. Here T_1 to T_n are standard Coq types, and P and Q are Coq predicates in our assertion logic; typically the model of each class will contain the building blocks required to construct these predicates.

Specifications can also mention the program variables that each method takes as arguments, and the this pointer, but making this compatible with the predicates we defined for the program model is not entirely straightforward. Program variables are strings, and their values are stored on the stack. The List_rep predicate has the type val → list val → asn. We would like to be able to declare an assertion List_rep "p" *xs* stating that the program variable p points to a list *xs*, but this will not type check in Coq as "p" has the type string and not val. In Charge! this problem is solved using an environment monad [11] that allows assertions to be parametrised by a stack, and all program variables to be evaluated before being used in a predicate. We will not go into the exact details here, but by prefixing any Coq predicate and its arguments with a backtick (`), we allow these predicates to use the stack to look up the value of any program variable. We also have the expression "p"%V that returns a function that, given a stack, looks up the program variable "p" on the stack. The List_rep predicate is then written as `List_rep ("p"%V) (`*xs*), where the parentheses can be omitted. Note that `*xs* is evaluated to *xs* since *xs* contains no program variables, but the backtick is still required to make the term type check. One important observation is that we can apply the environment monad to any Coq type, which allows us to use the standard built-in libraries of Coq with our specifications, even though they were not originally designed with program variables in mind.

The specifications for all methods in our list library are presented in Figure 7.4. The specification for nodeLength uses the Node_rep predicate directly since this method is defined in the Node class. The other methods are defined in the List class, and use the List_rep predicate for their specifications. A user of this library would only need to know about the List_rep predicate.

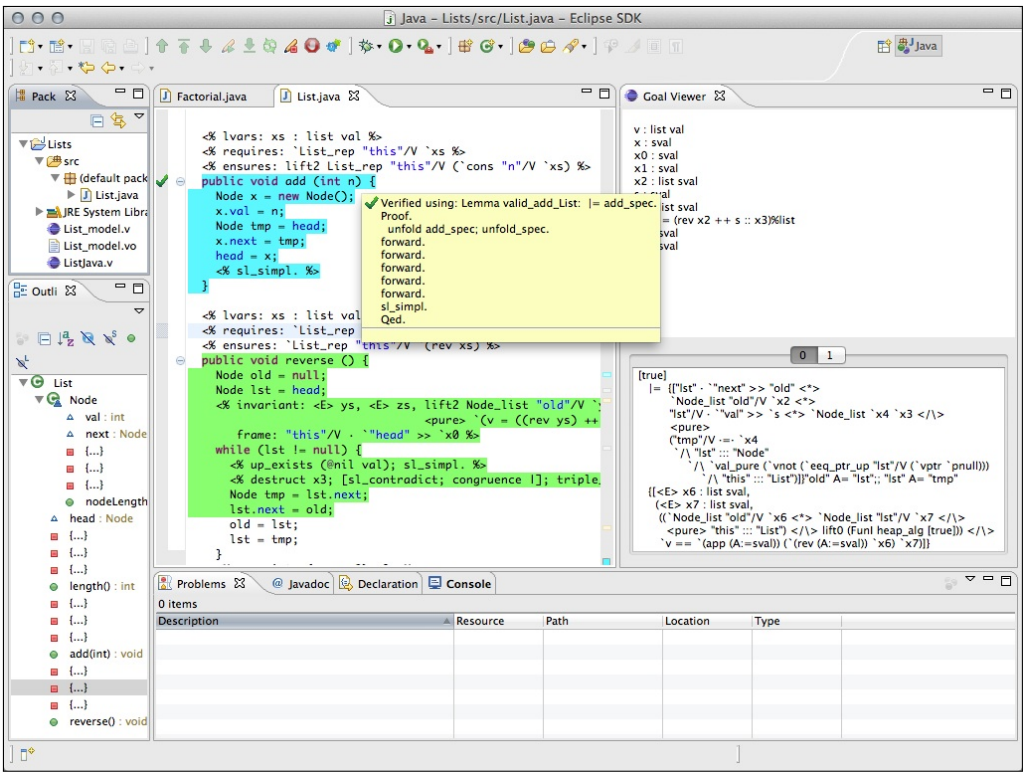


Figure 7.5: Screenshot of the Java perspective. In the middle, the Java editor with antiquotes for specifications and proofs. The green background indicates commands processed by Coq. The blue background indicates a verified method; its tooltip contains the lemma statement and its proof, including the calls to the forward tactic. To the right, the goal viewer, displaying the Hoare triple of the remaining loop body.

Formal Verification

Classes are verified method by method. By right-clicking on a method and selecting Verify Method from the menu, we enter a mode that lets us prove formally that the method satisfies its specification. A screenshot is presented in Figure 7.5. Charge! includes a semantics for Java that comes with an inference rule for every statement type. Methods are proved correct by stepping through them one statement at a time, using a tactic named forward, which is inserted automatically for each statement. The forward tactic checks whether the precondition is satisfied, and if so executes the next command symbolically and updates the program state. If the precondition is not satisfied, the user can insert standard

Coq code in antiquotes between statements to change the state in such a way that the next step is enabled.

The Java perspective uses the same goal viewer as the Coq perspective, and displays a Hoare triple $\{P\}\ c\ \{Q\}$ where c is the remaining statement, and the precondition P is the current program state. The predicate Q is most often the postcondition of the method, but it is also used at the end of loops and conditional branches.

Loops are annotated with two predicates. The first is an invariant, which is a predicate that must hold on loop entry and on all iterations of the loop. The second is a frame, which is a predicate that is part of the current proof state and which is not required to prove the loop body, but which is required to prove the remaining code of the method. The frame is only available outside of the loop. This is achieved by using to the frame rule from page 56.

The proofs of the different methods in the list library vary in complexity. The simplest one is the `add` method, which requires only its specification and a single `sl_simpl` annotation. The proofs for the `nodeLength` and `length` methods are relatively straightforward. At each method call, the precondition of `nodeLength` must be established, but the fact that the method is recursive makes no difference to the complexity of the proof. The most complex proof is the one for the `reverse` method, which is presented in Figure 7.6. This proof nicely demonstrates the way software is verified using Kopitiam. Manual proofs are required in key places, but these can often be kept small and concise. In particular, any properties pertaining to the model should be proved as separate lemmas in the model file.

Proof Certificate

For each method a validity theorem is produced, which states that the implementation fulfills its specification. Once this theorem is discharged, the method in question is marked with a green checkmark and light blue background, as shown Figure 7.5.

Once all methods of a class have been verified, Kopitiam generates a proof certificate. This certificate contains the Java program code as Coq definitions for all methods and classes. The theorem that the entire program is correct is derived automatically from the theorems that state

```
void reverse () {
  Node old = null; Node lst = this.head;
  <% invariant: ∃ys zs. `Node_list `"old" `ys ∗
                        `Node_list `"lst" `zs ∧
                        `(v = ((rev ys) ++ zs))
      frame: "this"/V.`head ↦ `x0 ∧ `typeof "this"/V `List %>
  while (lst != null) {
    //Proof of the loop invariant for loop entry follows
    <% up_exists (@nil val); sl_simpl. %>
    <% destruct x3; [sl_contradict; congruence |]; triple_nf. %>
    Node tmp = lst.next; lst.next = old; old = lst; lst = tmp;
  }
  <% ... Proof of the loop invariant for the loop body
        (3 lines) ...%>
  this.head = old;
  <% ... Proof of the postcondition (3 lines) ...%>
}
```

Figure 7.6: Proof of the reverse method. User guided proofs are required to prove the loop invariant for loop entry and for the loop body, and to prove the postcondition. Coq automatically introduces new logical variables for the binders we use. The logical variables v in the invariant represents xs from the specification, and $x0$ in the frame represents h from the definition of List_rep (page 57).

that each individual method is correct. Ultimately, we obtain one theorem of program correctness which can be checked completely by the Coq kernel using Charge! independently of Kopitiam.

7.3 Implementation

Kopitiam is not a stand-alone application, but an Eclipse plugin. Eclipse is written in Java, and thus runs on the JVM. Kopitiam is developed in Scala, a functional object-centered language which also runs on the JVM. In this section, we discuss two important aspects of our development. The first is how Kopitiam communicates with Coq and stays responsive while Coq is working on a proof. The second is how we extend Java's syntax with antiquotes while maintaining support of the Eclipse tools that work on regular Java projects. We will also briefly discuss parsing of Coq code, how we translate Java into Coq definitions, what measures

we take to increase performance, and finally how the Verify Method action is implemented.

7.3.1 Coq Interaction

Coq does not provide a standard API. All communication is handled via the `coqtop` binary, which in turn communicates solely via the standard input, output and error streams. Thus all data must be sent and received as strings.

The main obstacle to overcome when interfacing with Coq is to achieve a responsive and reliable communication. Coq often takes a long time to process commands and it is important that Kopitiam does not freeze while Coq is working. Moreover Coq only responds to complete commands, and care has to be taken to avoid a state where Coq waits for more input, while Kopitiam waits for a response from Coq.

We solve these problems by designing an asynchronous publish/subscribe system using Scala actors, shown in Figure 7.7. This allows users to send commands without the IDE being blocked while waiting for a response. Kopitiam has the central message distributor PrintActor that maintains a set of subscribers. The PrintActor receives a message as a string from Coq, parses it to a more structured form, and redistributes this parsed message to all subscribers. Kopitiam encapsulates communication with Coq within the CoqTop singleton object. This object provides a single method `writeToCoq`, which takes a string and transforms it to a complete command (by adding the terminating '.' if necessary) before sending the command to Coq.

The subscribers of the message distributor are shown on the right of Figure 7.7, we describe them in ascending order. The DocumentState object tracks Coq's state. Coq maintains a unique number for each successful state update, and DocumentState stores a mapping from a position in the source file to Coq's state. This makes it possible to step back an arbitrary number of commands to a previous state. The GoalViewer updates the goal viewer in the Coq and Java perspectives; the CoqEditor colors the background of the processed Coq commands and manages error reporting. The ContentAssistant stores names of proven lemmas in the current file and uses them to suggest completions to the user. The ContentAssistant also uses Coq's `SearchAbout` command to find possible completions

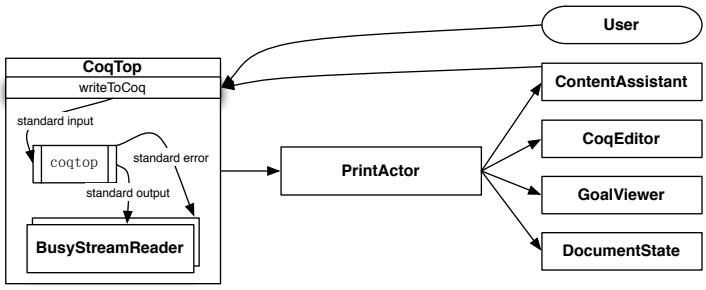


Figure 7.7: The singleton object CoqTop encapsulates the communication with `coqtop`. The method `writeToCoq` sends a message to the standard input stream of `coqtop`. The CoqTop singleton starts a process and connects the standard output and error streams to a BusyStreamReader instance each. When a BusyStreamReader instance receives a message, it forwards that to the singleton object PrintActor. This is the central distributor of messages, and first parses the received string into a more structured object and sends that asynchronously to all subscribers on the right side.

from other files. Finally, users interact with CoqTop every time they step through a proof or program.

This design for interaction with Coq is extensible. The Java perspective in Kopitiam has a lot in common with the Coq perspective; the user performs actions such as stepping through code, writing proof scripts, and retracting proof state. Nearly all code for communicating with Coq has been reused between the two perspectives. The only perspective-specific code handles the positioning information in the DocumentState object, since the syntax of Coq files and Java files differ.

7.3.2 Antiquotes

We use antiquotes to extend the Java syntax to handle method specifications, which are written before method definitions, and proof scripts, which are written inside method bodies and interleaved with Java statements.

In order to integrate these antiquotes with the Java syntax we extend the Java lexer and parser in Eclipse. Since neither the lexer, nor the parser, provide an off-the-shelf extension mechanism, we use aspect-oriented programming to modify the bytecode of Eclipse at load-time. Our extension creates a dummy node for each antiquote in the abstract syntax tree of the Java programs. The developer can still use all standard Eclipse tools like debuggers, compilers, profilers, and unit testers, since

the dummy nodes are ignored by Eclipse. The extensions to the Eclipse parser and lexer are available as a separate Eclipse plugin[1].

We evaluated three other possible approaches to embed specifications and proof script inside of programs: annotations, comments, and static method calls.

Java annotations are only allowed before methods, not inside method bodies, and would only support method specifications. Type annotations are a recent development that allow annotations at type occurrences, but since they also can not be placed inside the method bodies they do not solve the problem.

Java comments are allowed anywhere, making them a good candidate. However, the Eclipse Java lexer throws comments away[2]. If we used comments, we would have to modify the internal behaviour of the Java lexer (to keep the embedded statements) and parser (to make sense of these embedded statements). This would make our implementation prone to breakage with respect to future releases of Eclipse.

In an earlier version of Kopitiam [79] we represented method specifications and proofs in the Java source code as strings in calls to dummy static methods on a dummy `Coq` class. This was cumbersome and all commands had to be passed as strings. Moreover, method calls are not allowed outside method bodies, where specifications are written.

7.3.3 Parsing of Coq Code

Kopitiam needs to parse Coq code to support features like syntax highlighting, an outline, and proof folding. The folding hides proof script of a lemma between the `Proof` and `Qed` keywords, similarly to how method bodies are folded in Eclipse.

Rather than having a complete parser for Coq's grammar, we first split Coq code into sentences of complete commands. Each sentence is then parsed individually to check for the keywords that are required by

[1] `https://github.com/hannesm/KopitiamAspects`

[2] Update from July 2013: This paragraph is not entirely correct. There is no direct access to the comments via the `ASTNode` API (especially the `statements` method on a method body does not return comments), but the Eclipse AST contains comments, retrievable via `ASTVisitor` for example. The main argument against using comments is that they are unstructured and therefore do not enable further usability features, such as highlighting and completion.

the highlighter, the outline, and the folder. Parsing the entire Coq code is not realistic as Coq allows users to dynamically change its grammar on the fly using a powerful notation mechanism. Since even compiled Coq binaries can include such changes, creating an external parser for complete Coq commands is impractical. Our solution allows us to only check each command for the keywords that we require to build the functionality we need. We leave the complete parsing of each individual command required for proof checking to Coq.

7.3.4 Java Translation

Kopitiam translates Java into the subset of Java formalized in Charge!. This support includes read and write operations for fields and variables, static and dynamic method calls (including recursion), while loops, conditionals, and object allocation. The latter creates a new instance of an object and allocates memory for its fields on the heap.

The translation is implemented using the visitor pattern on the Eclipse abstract syntax tree. During the traversal, Kopitiam issues warnings about missing specifications and Java code which it cannot translate to our core subset.

7.3.5 Verify Method

The Verify Method action in the Kopitiam user interface requires that the theory file containing the program model has been compiled, and that Coq definitions have been created for all method specifications. After these things are done, a validity theorem for the method is generated and the user is prompted to interactively verify the correctness of the method.

To increase responsiveness Kopitiam is optimised to load theories, including the program model, only when needed and unloading them only when they are modified. Moreover, Kopitiam adopts a local translation technique that translates modified code and splices it into the already translated program. The whole code is retranslated only when major revisions are made.

7.4 Discussion and Future Work

The most challenging aspects of developing Kopitiam are done. The difficult part has been figuring out how to integrate the Java IDE of Eclipse with Coq, but that aspect is complete and working well. We improve significantly on our previous release [79] where Kopitiam only supported translation of Java code into Coq definitions which had to be verified completely in the Coq perspective. For this release, the Java perspective is entirely new and we are able to conduct proofs directly within the Java editor. The Coq perspective has also gained several key features such as proof completion, syntax highlighting and the proof outline.

There are things we wish to improve, both on the IDE side (for usability reasons) and in the back-end Charge! (for performance and expressiveness reasons). Currently, programs must fit into one file. The subset of Java we support is defined in Section 7.3.4. Additionally, these further restrictions are imposed on the language: only a single return statement at the end of the method body is supported; the primitive types must be integers, booleans, and object references; finally, class-to-class inheritance and interfaces are not supported.

Adding more language features is a two-step process. Charge! has to be developed to include the theoretical results required, and Kopitiam must be extended to allow easy access to these results. We have developed theories to reason about inheritance using interfaces [12], and we plan to add support for this in the near future. We are also making Charge! modular by adding support for unimplemented specifications and program composition. This will allow classes to be verified independently and then combined to form a complete program.

There are three aspects of the user interface that require attention. The first is the way we write specifications, frames, and loop invariants. Currently, they are passed as raw data to Charge!. We plan to support a domain specific language in the antiquotes, with proper syntax highlighting, that hides implementation details from the users, such as the environment monad, as seen in Figure 7.6, namely fresh names, backquotes and *N*. Secondly, Kopitiam generates fresh names when quantifiers are moved to the Coq context. These names are typically not the ones chosen by the user, but ones generated by Charge!, which can also be seen in Figure 7.6. Charge! uses higher-order abstract syntax

to handle binders, which makes it impossible to get a handle on their names in Coq. Two ways around this are either to change how binders are modelled in Charge! or to extend the functionality of Coq to allow these bound names to be retrieved. Finally, we plan to redesign the goal viewer in the Java perspective. Currently, both the Coq and the Java perspectives use the same goal viewer, and the user sees the goal as a complete Hoare triple, which is the internal representation of the program in Charge! (see Figure 7.5). The ideal is to only see the precondition of the triple, which is the current program state, properly split into views describing the heap, and views describing the stack. This will make working in Kopitiam much like using a debugger, where users step through the program and are presented with a clear view of the program state.

We still have a few performance issues. These are mostly related to the Coq back-end. Charge! handles the symbolic execution of commands as the user steps through the program in the Java editor, and this can be slow if the next command has a precondition that is difficult to prove. We plan to rewrite our heuristics using reflective tactics, similar to the ones used by Chlipala in his Bedrock framework [34], and this will speed up proof search considerably.

The largest case study we have verified using Charge! is the snapshotable trees library from the C5 library [82]. Currently, this formalisation only exists as a Coq theory and a next logical step is to certify the library using Kopitiam.

We are currently using Kopitiam in a master course on software verification based on the *Software Foundations* book [99][3] by Pierce et al. So far, student feedback has been very positive and the Coq perspective of Kopitiam is proving capable of dealing with all material and exercises presented in the book.

7.5 Related Work

Several tools for program verification of varying complexity have appeared in the last decades. Like in Kopitiam, source code is typically annotated with specifications, after which the tools check if these specifications are satisfied.

[3] `http://www.cis.upenn.edu/~bcpierce/sf/`

Many tools strive for automation, which by necessity limits their expressivity. Typically, they produce a large number of lemmas that are automatically discharged by an automatic theorem prover, such as Z3 [85] or SPASS [117]. Examples of such tools are Spec# for C# [7] and ESC/Java2 for Java [36]. A few tools also use separation logic, like Smallfoot [13], Space Invader [32], SLAyer [14], and jStar [45].

To verify full functional correctness of programs, interactive techniques are required. The options that have been explored have followed one of three paths. The first is to generate proof obligations from annotated programs, feed them to an interactive proof assistant and have the user discharge the proof obligations manually. The second is to write the program, its specifications, and the proof of correctness entirely in an interactive proof assistant. Finally, a development environment for verified software can be developed from the ground up, possibly hooking into external proof assistants as required.

The first approach is used by tools like Jahob [70] and Loop [113]. Loop translates JML-annotated Java programs to the proof assistant PVS, which is then used interactively to discharge the goals. Jahob uses its own specification language and uses both interactive proof assistants (Isabelle and Coq) and automatic theorem provers (e.g. SPASS, CVC3, and Z3) to discharge the goals. The disadvantage to this approach is that it is difficult to keep the proof script synchronised with the program – once the script has been generated, any changes to the program will not automatically propagate to its proof and there is a danger that proofs refer to old instances of the programs. Also, neither tool uses separation logic, making it difficult to reason modularly about programs with mutable state.

The second approach (programming directly in a proof assistant) has had great success. Two notable examples are the verified L4 microkernel by Klein et al. [64] and the verified optimising C compiler CompCert by Leroy [75]. These two projects have set the standard for state-of-the-art formally verified software today, but each also required a Herculean effort to complete. Both projects verify around 7000 lines of code, yet their proof sizes are orders of magnitude larger (around 50000 lines of code for CompCert and 200000 for the L4 kernel); the L4 kernel required around 25 man years to complete, CompCert around 10. For the L4 kernel, the Simpl library by Schirmer [102] was used to translate C

code to Isabelle. The CompCert compiler is written in the functional language Gallina, which is used inside Coq, and then extracted to an executable OCaml program.

The third approach (building the development environment from scratch) has been used for designing development environments such as KeY [1], Verifast [58] and Why3 [23]. KeY and Verifast have both been used successfully to verify Java and JavaCard programs [84, 97]. They both use first-order logics (Dynamic logic for KeY and separation logic for Verifast), and they both use custom-made IDEs and interactive verification environments. Verifast integrates Z3 to discharge many proof obligations automatically. Why3 certifies programs written in the functional programming language WhyML, which can be extracted to executable OCaml code. It is the only one of these three tools that hooks into a general-purpose interactive proof assistant (Coq) to allow users to discharge proof obligations interactively. Why3 also hooks into a wide range of automatic theorem provers to help discharge proof obligations automatically. The advantage of this approach is that a lot of work is done automatically by the tool. The disadvantage is that these external provers are treated as trusted oracles which dramatically increases the size of the code base we have to trust as we have to assume that these tools are bug-free, as opposed to trusting only the Coq kernel. The Sledgehammer tool for Isabelle by Blanchette et al. [21] is designed to hook automatic theorem provers into interactive ones in a safe manner by providing techniques for the interactive proof assistant to replay the proofs generated by the automatic ones, but these techniques have not yet been implemented in Coq.

The work that mostly resembles ours comes from the Mobius project and Mobius PVE [9], which integrates several verification environments for Java with Eclipse. It includes the JACK [8] framework that takes JML-annotated Java code, generates proof obligations using a weakest precondition calculus, and discharges these proof obligations either with automatic or interactive proof assistants, including Coq. Mobius PVE comes with the ProverEditor (previously called CoqEditor) perspective for Eclipse that, like Kopitiam, allows users to develop Coq theories [33]. By using this perspective, JACK keeps the whole development cycle in Eclipse.

Comparison

Kopitiam sets itself apart from all other tools in that it is based on separation logic, it is hooked up to a general-purpose interactive proof-assistant, and it allows programs to be written and verified in an industry-grade IDE. All these points are important. Tools not based on separation logic are unlikely to scale to larger programs; moreover, extending general-purpose well maintained frameworks like Coq and Eclipse, both of which are arguably leaders in their respective fields, is preferable to building custom-made versions of our own; designing any of these two systems from scratch would be full fledged research and engineering projects in their own right.

Of the interactive tools listed above, only Verifast is based on separation logic. Jahob, JACK and Loop all generate proof obligations for general-purpose proof assistants but these are all stored in separate files and not actively kept synchronised with the code. Only JACK uses Eclipse, whereas Verifast, KeY and Why3 use their own custom-built IDEs.

7.6 Conclusions

Kopitiam provides the closest integration to date between an industry-grade IDE and a general-purpose proof assistant. The ability to step through the source code of a program when developing its proof is instructive, and provides an intuitive interface for programmers and proof developers alike. Moreover, Kopitiam provides a development environment for Coq theories where users can define program models, postulate lemmas and theorems, and prove these directly in Eclipse. Kopitiam also generates certificates, checkable by Coq, for each verified program. Finally, since we use an expressive higher-order separation logic we are able to model both the semantics of the Java programming language as well as the Java programs themselves in Coq. This means that the proof certificates are unified theorems, checkable by the Coq kernel, that guarantee that a verified program follows its specifications with respect to the operational semantics of our model of Java. This ensures that the trusted code base is kept very small. The only two things we have to trust are the model of Java in Charge! and the Coq kernel.

Acknowledgement

Thanks to Peter Sestoft, Joseph Kiniry, Lars Birkedal, Fabrizio Montesi, Jasmin Blanchette, and Filip Sieczkowski for their valuable feedback on this paper. We want to thank David Raymond Christiansen, Alec Faithfull, Daniel Dam Freiling and Mads Hartmann Jensen for contributions to the Kopitiam source code. Thanks to Stephan Herrmann for helping out with the Eclipse parser and lexer internals.

Chapter 8

Evolutionary Design and Implementation of Kopitiam

Abstract

We present the design and implementation of Kopitiam, an Eclipse plugin for integrating development of Java programs with the development of full functional proofs of their correctness using the proof assistant Coq. First, we motivate the workflow of a potential user of Kopitiam and the requirements imposed by the intended workflow. While literature commonly treats design and implementation statically, we review and critically analyse the evolution of Kopitiam from a software engineering perspective. We focus on the lifecycle of Kopitiam itself, which consists of design, implementation, detection of shortcomings, adjustment of design, adjustment of implementation, finding of more shortcomings, and so on.

Furthermore, we will describe the impact that the external software dependencies, Eclipse and Coq, had on Kopitiam. In particular, we will analyse how new features of Eclipse or Coq permitted or required adjustment of Kopitiam's design.

8.1 Introduction

The ultimate goal of the research project "Tools and Methods of Scalable Software Verification"[1] is to support mechanized full functional verification of Java programs. To bridge the gap between developing Java programs and their proofs of correctness, we have designed and incrementally developed Kopitiam.

Kopitiam is a plugin for Eclipse to interactively prove correctness of Java programs using separation logic and the proof assistant Coq. Proofs are written side-by-side with developing Java programs in Eclipse, allowing seamless integration into the developer's workflow.

In this paper we describe the design and implementation among the different stages undergone by Kopitiam during this research project. We distinguish four generations of Kopitiam's design and implementation, each of which consists of multiple releases. The first generation (0.0.*) of Kopitiam is described in more detail in an earlier paper [79]. Details of the second generation (0.1.*) have not been reported in a scientific publication as it had poor performance, which were addressed in the third generation (0.2.*). The more recent design and implementation of the third generation is presented in a technical report [81]. The remainder of this paper will present the current stage of Kopitiam, which is the fourth generation (0.3.*).

In Section 8.2 we recapitulate Lehman's classification of software [73]. In Section 8.3 we propose a software development workflow which integrates verification. We also describe the workflow for developing proofs in Coq. In Section 8.4 we describe the main contribution of this paper, and in Section 8.5 a set of requirements for Kopitiam and solutions to implementation challenges. We conclude in Section 8.6 and describe future work in Section 8.7.

8.2 Background

To understand software evolution, we recapitulate a classification of software, which Lehman [73] developed more than 30 years ago. This classification distinguishes three kinds of programs, so called S, P, and E programs.

[1] `http://www.itu.dk/research/tomeso/`

An S program is a program whose functionality is formally defined by and derivable from a specification. Examples of S programs are sorting a collection or reversing a list. The implementation of an S program is finished if it complies to the specification. Modifications for maintenance, such as improving code clarity can be applied, but modifications that alter the behaviour cannot. S programs are the building blocks of bigger programs.

The real world contains uncertainties and unknowns, which gives rise to P programs. These can no longer be clearly specified without considering the world and the context of use. P programs contain an intrinsic feedback loop: the acceptance of a solution of a P program is determined by the environment in which it is embedded. The crucial difference between S and P programs is that judgements about the correctness of S programs relate to the specification, whereas in P programs such judgement is obtained in its real-world context. An example of a P program is a weather forecast program: deeper understanding of the weather results in more accurate algorithms for the forecast, to which the P program needs to adapt. P programs must also change when the world around them changes. All useful P programs undergo never-ending changes, and are never "finished".

The last class of programs are E programs, which mechanize a human or societal activity. The program itself becomes part of the world it models. An example is an operating system; given that technology improves and new hardware is manufactured, this new hardware has to be supported. Additionally, advertising influences society, and people's expectations of the new hardware and operating system rises. P and E programs are closely related; the difference is that an E program influences its own requirements.

Kopitiam uses parts of the constantly changing world, because it builds on other software. We consider Kopitiam to be either a P or E program. Considering Kopitiam being an E program is due to the fact that user expectations feed back on its requirements. Kopitiam undergoes never-ending changes, because use gives rise to suggestions for improvement.

Although Kopitiam supports modification of specifications and programs, the programs that have been developed and verified in Kopitiam to date are of type S. There is no inherent technical limitation in Kopitiam which prevents users from verifying those parts of P or E programs

which can be formally modeled, but so far no case study has been conducted with a P or E program. It is difficult to capture a full specification of both P and E programs, because mathematical models are not suitable to describe these.

8.3 Software Development and Software Verification Workflow

From the previous discussion it is clear that software development consists to a large extent of software maintenance. Already Lehman [73] discovered that more than 70% of the software development budget is spent on maintenance, whereas only 30% is spent on development. Brooks [28] gives more evidence that a huge amount of time is spent on maintaining software, rather than developing software.

To understand how software is maintained, we consider the common programming workflow. A common programming workflow is that a developer first writes a program (edit), then uses a compiler to translate it into machine code (compile), afterwards executes it (run) and finally debug it (debug). The program itself is most likely a library, and the execution thereof is rather the execution of unit tests. If the requirements for the program change or it needs modifications for maintenance, the presented programming workflow is followed to adapt the program to meet the modified requirements. Additionally, most object-oriented software is developed with the help of an integrated development environment like Eclipse, which provides tools such as compiler, debugger, profiler, navigation, documentation, unit testing.

To be maintainable and reusable by others than the initial author, a program needs to be documented well and its documentation must be up to date. When a program is modified, its documentation should reflect these modifications. If the documentation lives in a separate location, such as a technical report or powerpoint presentation, the probability that it gets updated is very low. To ease the burden on the developer, Knuth invented literate programming [65], in which the developer writes documentation and embeds the program in it. The big advantage is that the program and its documentation are synchronized and maintained together. Compilation of the program requires to first split it off from the documentation. A less radical approach is represented by Javadoc, a documentation system for Java where the documentation is

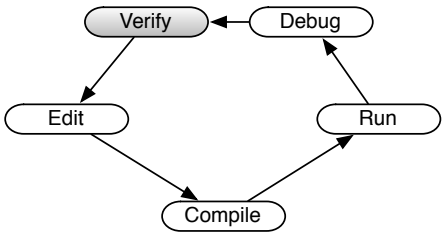


Figure 8.1: The common development workflow - edit, compile, run, debug - is extended with the additional step verify.

embedded into a Java program using specific comments. The advantage of Javadoc over literate programming is that the program can be compiled and analyzed with off-the-shelf tools. Furthermore, Javadoc uses static information to produce hyperlinked documentation files, which are easily readable with a web browser.

We believe that, similar to documentation, correctness proofs of programs are important, and they also need to be maintained together with the program. Only if we integrate verification into the software development workflow and the software development tools, we will get maintained verified software. We add *verify* as another step to the programming workflow, as shown in Figure 8.1. To support this modified workflow, we present Kopitiam, an Eclipse plugin which integrates development of Java programs with verification of their correctness. The burden on a developer is reduced: she does not need to switch the environment to adjust correctness proofs or modify the program code.

We describe the detailed workflow to interact with the proof assistant Coq, which is followed by a detailed description of the workflow to develop correctness proofs of Java programs.

8.3.1 Workflow: Coq Interaction

The interaction of developing proofs in Coq is similar to programming in BASIC or Pascal. Definitions must be provided before they are used, and lemmas can only be used after they have been proven. A Coq file is processed one command at a time, from top to bottom. This is in contrast to many modern programming languages, which do not require a developer to write definitions in dependency order. A Coq file consists of definitions in the purely functional programming language Gallina,

and proofs. A proof is a lemma statement followed by application of tactics, the so-called *proof script*. The proof script produces a proof term with holes. Each hole corresponds to one proof obligation. A user interface element displaying the current proof obligations and their contexts is necessary for a user who interactively discharges proof obligations.

To interactively develop the proof, a user steps forward and backward in the proof script and looks at the proof obligations, which are updated after each command. The user needs some visual feedback in the editor to see which parts of the proof script have already been processed, and errors in the proof script, both syntactic and semantic, need to be reported. If a user modifies an earlier, already proven lemma, the user interface needs to retract to the earlier definition and re-prove the lemma.

When interfacing with Coq, we have to keep in mind that the time that Coq needs to process a single command is hard to predict, and can be high (loading a theory can easily take several seconds). To keep the user interface responsive, interaction with Coq must not block the user interface, and the number of commands sent to Coq should be minimized.

There are two actively used development environments for Coq: Proof General [5], which is a mode for the editor Emacs; and the GTK-based user interface CoqIDE, which is distributed together with Coq. Kopitiam's unique characteristic is a tight integration of the proof and software development environments, and we chose Eclipse as our Java integrated development environment. Thus we needed to develop also a Coq development environment as an Eclipse plugin.

8.3.2 Workflow: Java Proof Development

Now that we have described the Coq proof development, we are ready to describe the workflow required to prove correctness of a Java program. We first develop a model for the Java program inside of Coq's pure functional programming language Gallina. Then we develop *representation predicates*, which bind the model to the Java program. A representation predicate receives a reference from the Java world and a corresponding model. It emits a formula, which describes the heap shape. The formula consists of separation logic predicates, using standard logic operators: conjunction, separating conjunction ($*$), and the points-to predicate

($\mapsto$). We use these representation predicates to write method specifications in Hoare style with a pre- and postcondition. Finally we verify interactively that each method conforms to its specification by using our formalised Java semantics [12].

Let us consider a Java program, the class List, which implements a singly linked list containing Node elements (lines 2–4). The single method add (lines 9–15) inserts a Node with the given integer value at the front:

```
public class List {
  static class Node {
3   int value;
    Node next;
  }
6
  Node head;

9 public void add (int n) {
    Node x = new Node();
    x.value = n;
12  Node tmp = head;
    x.next = tmp;
    head = x;
15 }
}
```

Next we need to develop a mathematical model of lists. The model has to be convincing to a reader, because all we verify is that the Java implementation corresponds to the model. We can also develop lemmas that describes properties of the model, which are used for the Java correctness proof.

The model for our `List` class is an inductive `list` data structure in Gallina. It consists of two constructors, `nil` and `cons`. Our model is a finite sequence, whereas the Java class Node could potentially be cyclic. The Java insertion method (add) prevents a user from constructing such a cyclic list.

```
Inductive list (A : Type) : Type :=
  | nil  : list A
  | cons : A -> list A -> list A.
```

To continue our example, we define the representation predicate Node_list for the Node class and the representation predicate List_rep for the List class.

The Node_list predicate is defined recursively, as a Coq fixpoint. If the model lst is nil, the reference p must be null (line 3). In the recursive case, p has the dynamic type Node (line 4). The reference to the head of the Java list, p.value, points to a value which is equal to the head of the model, x (line 4). The next field points to some v, for which the Node_List predicate holds (line 5), with the tail xs of the modelled list.

```
Fixpoint Node_list (p : val) (lst : list val) : asn :=
  match lst with
3   | nil       ⇒ p = null
    | cons x xs ⇒ typeof p Node ∧ p.value↦x ∗
                  ∃v : val. p.next↦v ∗ Node_list v xs
6 end.
```

The List_rep representation predicate receives a reference p of type val, and a list of values lst. The dynamic type of p is "List", there exists a reference h, and the head field of p points to h. The Node_list predicate holds for h and the given model lst. The reason to have both a list representation predicate and a node representation predicate is that they have each a corresponding class in the Java program, and it allows us to distinguish the empty list from the reference null.

```
Definition List_rep (p : val) (lst : list val) : asn :=
    typeof p List ∧ ∃h : val. p.head↦h ∗ Node_list h lst
```

We use the representation predicates to formulate the method specifications by writing pre- and postconditions. A precondition contains the assumptions, which are the predicates that must hold when this method is called. The postcondition contains the promises which hold when the method returns. Together they form a contract [83]: before the method can be called, the precondition must hold. When the method returns, the postcondition holds.

Our example contains only a single method, add. The contract is that for any list xs, the assumption List_rep this xs holds. The postcondition

promises that n will have been prepended to the list. We write this using a Hoare triple:

$$\forall xs.\ \{\ \mathsf{List\_rep\ this}\ xs\ \}\ \mathsf{add(int\ n)}\ \{\mathsf{List\_rep\ this}\ (\mathsf{cons\ n}\ xs)\ \}$$

Finally, the correctness of each method is proven separately. The currently valid predicates describe a *symbolic heap*. The method precondition is assumed initially: thus, the symbolic heap consists of the precondition. For each statement of the method body, the corresponding semantic rule is applied. The semantic rules are defined in Bengtson et al. [12] and each consists of a Hoare triple: the precondition, the statement itself, and its postcondition. To apply a rule, the current symbolic heap has to satisfy the statement's precondition. Predicates not used by the rule can be ignored using the frame rule of separation logic. The postcondition of the rule specifies the modifications to the symbolic heap. At the end of the method body, we need to verify that the symbolic heap satisfies the method postcondition.

Once we have verified that all methods conform to our specification, we have proven the program correct.

8.4 Requirements

We faced three additional requirements for development of Kopitiam: (1) we have limited developer resources; (2) proofs developed with Kopitiam should be accepted by the verification community; and (3) it must work on any major platform (Windows, MacOSX and Linux).

To address (1) and (3) we reuse available and actively developed software, Coq and Eclipse, instead of implementing our own proof assistant and integrated development environment for Java.

We addressed (2) by using an off-the-shelf trusted proof assistant. A common approach for proof assistants, named LCF, is to be based on a small trusted core, which is generally accepted. We are using the interactive proof assistant Coq [111]. Coq is based on the calculus of inductive constructions, is available as open source software, has existed for decades, has a large user community and is actively developed. Several large projects use Coq, for example CompCert [75], which is a semantics-preserving optimizing compiler for C programs, written and

verified in Coq. Gonthier used Coq to prove the four color theorem [50], and more recently also the odd order theorem [51].

In our research project we developed Kopitiam side-by-side with Charge! [12, 11], a formalisation of higher-order separation logic and an encoding of semantics for a subset of Java within Coq.

To address (1) and (3) we are building Kopitiam on top of Eclipse, an open source integrated development environment initially developed for Java programming. Eclipse has a powerful plugin mechanism, a massive and well-documented API for extensions, a large user community, and is actively developed. Using Eclipse has several advantages: the common features of Eclipse, such as compilation, testing, execution, debugging, and profiling are already available; developers familiar with Eclipse do not have to learn a new development environment; and enhancements to Eclipse are immediately available in Kopitiam.

Kopitiam is designed to accommodate all steps of the presented workflow of verification of the correctness, and a user should not need to switch to a separate environment to complete a proof because some feature is missing. The common user expectations of development environments should be met; also the common expectations that programs should be robust and responsive. Kopitiam should always allow the user to edit programs and cancel running jobs.

The verification steps described in the workflow for proving correctness of Java programs can be classified into two categories: the interactive Coq development, in which the model and the representation predicates are developed; and the Java integration, where the Java program is extended with method specifications and proof scripts, which is also done interactively.

Both categories have different requirements, which we consider separately. They are also separate perspectives of Eclipse: while we developed the Coq perspective from the ground up, we only needed to extend the already existing Java perspective of Eclipse to accommodate method specifications and proof scripts.

The Coq perspective needs to meet users' expectations for a proof assistant frontend, similar to Proof General or CoqIDE. Among these expectations are:

- A user interface element that displays the current proof obligations.

- Visual feedback in the editor for the commands that are already processed.
- Visual feedback for the commands that Coq is currently processing.
- A set of buttons and keyboard shortcuts to invoke actions, such as stepping forward, backward and to the current cursor position.
- Errors and warnings from Coq should be reported.
- The possibility to interrupt Coq while it is processing a command.
- When the user edits an already processed command, Kopitiam must retract the processed region to before the command, in order to keep the state in Coq and in the editor in sync.
- Code completion, which is more common in integrated development environments than in proof assistants, should be supported. If a user starts to type, possible completions are suggested. Similarly, during development of Coq code, the available and usable tactic names should be displayed dynamically.
- An outline view for easy navigation within a single file is necessary to keep track of larger developments.
- Automatic builders and dependency management, which attempt to re-prove dependent theories on successful modification of another one.

The resulting perspective does meet these, and can be used also for general Coq proof development, not necessarily related to Java proofs.

The Java extensions should allow a user to develop a correctness proof of the Java program. This leads to several requirements:

- A user should be able to annotate Java programs with specifications.
- Annotating Java programs with proof script should be supported.
- Stepping through the proof script has to be possible.
- The symbolic heap should be visible when proving a method.

- Error reporting of either specifications or proof script should be precise.

Behind the scenes, Kopitiam has to translate the Java program into a Coq representation of Java's abstract syntax, and update these definitions whenever the Java program is modified. To react properly to modifications of a method specification, Kopitiam has to use the call dependency graph, and invalidate the proofs of callers of the modified method specification.

8.4.1 Requirements for the Implementation Language

The main constraint on Kopitiam's implementation is the need to run on the Java virtual machine (JVM) and integrate with Eclipse, which is written in Java.

Java itself is very verbose, in part because it does not include any type inference. Also, Java has several non-intuitive design decisions: type erasure, checked exceptions, and a non-unified type system. We looked for more succinct alternatives, and chose Scala, a functional programming language on top of the JVM. Scala cleans up some of Java's design decisions and adds a number of other features which allow us to write more concise and expressive source code.

Java integration is easy with Scala. The concurrency model with actors and immutable messages was also an attractive factor, as was the ability to do pattern matching.

Scala [92] provides a type system with local type inference, which reduces the typing burden. Scala also integrates higher-order functions and anonymous functions with a concise syntax. Scala provides powerful implicit conversions to develop type conversions whose scope can be contained.

Scala also provides traits, which are basically Java interfaces that can contain program code, but allow multiple inheritance. This turned out to be very useful for sharing functionality between the Java and the Coq perspectives. An example is the Coq interaction.

8.5 Implementation Challenges

In this section we describe several challenges and how we approached these. Being a P program, both the requirements and the potential solutions change over time, because we also use other software. In the following, we will discuss problems with implemented and discarded solutions in Kopitiam. In particular we will see how newer releases of the software used by Kopitiam, Eclipse and Coq, included some useful features which led to redesign of Kopitiam. Some Coq releases changed the behaviour slightly, and we had to adapt Kopitiam to cope with these changes.

Some solutions looked good in theory, but were impractical due to too high processing time. We focus on some challenges in this section, each of which we overcame using different approaches.

8.5.1 Communication with Coq

Kopitiam uses Coq as an interactive proof assistant. The commands are sent to Coq, which in turn sends back the current proof obligations with contexts or reports an error.

Coq is written in OCaml, and unfortunately there is no documented API to interface with Coq. In 2010 we looked into two approaches: either call OCaml code from the JVM natively, or to use the same way Proof General [5] interfaces Coq. Proof General runs the binary `coqtop` as a separate process and screen-scrapes its output streams.

There are some bridges to call OCaml code from Java[2], but when using these, the OCaml type information is lost.

Coq does not contain an API and furthermore is not available as a shared library, and, in contrast to Java and the JVM (and thus Kopitiam), OCaml code is platform-dependent. So, in order to call Coq code directly, we would have to distribute custom builds of Coq with Kopitiam. Thus, we would need to maintain a customized branch of Coq. As mentioned in section 8.4, our development resources were limited. We could also stick to a specific Coq version, and not upgrade past that version, but this approach would make it harder to evolve Kopitiam. Another issue with a non-standard Coq would be fewer trust.

[2]for example `http://ocamljava.x9c.fr/` and `http://forge.ocamlcore.org/projects/camljava/`

We were left with the second approach, to screen-scrape the output streams of `coqtop`, in the same manner as Proof General. The binary `coqtop` has support for an `-emacs` option, which outputs some numeric state identifiers. These state identifiers simplify the distinction between successful and erroneous commands.

Before Coq 8.4 these were the only possibilities. In the first three generations of Kopitiam the output of `coqtop` is screen-scraped. The screen-scraping code in the third generation of Kopitiam is almost always robust. However, sometimes due to buffering, remaining parts of the proof obligations are treated as unknown output instead of being part of the proof obligation.

Output from Coq is interleaved on the standard output stream and the standard error stream in an unspecified manner. Therefore, we screen-scrape both standard output and standard error in separate threads. We cannot reliably match a response to a specific request, because we do not know about timing behaviour. We use a publish-subscribe system to distribute the output of Coq to all interested components. Due to this, we have a technical restriction of a single `coqtop` process for Kopitiam. This restriction implies that when the user focuses a different editor and wants to conduct a proof inside of that editor, we have to carefully retract the current state of Coq.

Another problem is that the output produced by Coq is not exactly the same in all versions of Coq. In some versions hardcoded strings change, so we need to keep our code up to date with new Coq releases.

Coq version 8.4 supports a more robust interaction method. When `coqtop` is run with the `-ideslave` option, it accepts structured input and output in XML format. The XML format is not documented, but the OCaml code, that implements it is readable and the protocol syntax has not changed in recent releases (Coq 8.4, 8.4pl1 and 8.4pl2 use the same syntax). There are two big advantages to this structured format: only one stream is used, so we do not need multiple threads which read different streams, and we can easily match the response to a previous request. The XML interface is used by recent versions of CoqIDE as well. We developed Kopitiam (generation 0.3.*) to use the structured XML interface. Since its fourth generation Kopitiam interacts with multiple Coq processes at once, using one for each editor.

The evolution of Coq, towards a structured input and output format, is evidence that Kopitiam is a P program, and its evolution is largely driven by changes in the world, rather than in Kopitiam itself.

8.5.2 Embedding Specifications and Proofs in Java

Kopitiam integrates program and proof development. To keep the program and the proof in sync, it must embed the proof into the program. All existing Eclipse tools, such as execution of the program, unit testers, debuggers, profilers, and so on, have to work flawlessly with the program that has a proof embedded.

Both the method specification and arbitrary proof scripts must be embeddable into the Java program. To enhance readability of method specifications, they should be close to the method definition. The proof scripts should be interleaved with statements in a method body, to be directly connected to the statement whose proof obligation the proof script discharges.

We considered the following five different approaches to embed specifications and proof script:

1. Write them inside of comments
2. Calls to static methods.
3. A combination of comments and calls to static methods.
4. Write them as Java annotations.
5. Invent custom antiquotes.

Out of these five, we started to implement the second in Kopitiam's first generation. The specification and proof script was passed as a Java string, which did not allow for customized completions, syntax highlighting, and so on. Our experience showed that we want the embeddings to be more structured, but neither the comments nor the calls to static methods allowed to be more structured.

We evaluated Java annotations (Option 4) again, but the places where annotations are allowed in Java code are too limited for proof scripts, which are allowed between statements.

Since its second generation, Kopitiam implements antiquotes. We extend Eclipse's lexer and parser to accept <% and %>, between which we embed the specifications and proof scripts. Eclipse does not provide an API to extend its lexer and parser, so we instead use aspect-oriented programming to modify the code at load time. These antiquotes are present in Eclipse's abstract syntax tree, but the common tools like compilers, debuggers, profilers, unit testers, and so on, ignore these unknown nodes.

The approach of using aspect-oriented programming turned out to be robust and fast. A major upgrade from Eclipse Helios (version 3.6) to Juno (version 4.2) did not require any modification to the aspect-oriented code.

Although other software evolves around Kopitiam, a program of type P, it survived even major software upgrades. This will not universally be the case, but we designed Kopitiam to contain only a minimal amount of internal knowledge about external software.

8.5.3 Java Parser

Kopitiam needs to convert the Java program into a Coq representation. To achieve this, Kopitiam parses the Java program and emits a simplified syntax based on s-expressions. This simplified syntax does not handle complete Java code, such as nested field accesses, but many Java programs are easily refactored to the Java subset that our Coq formalisation supports.

There are two practical ways to get a parse tree of a Java program: either write a new Java parser, or use the existing one in Eclipse. In the first two generations we used the former approach, based on Scala's parser combinators, so that we could restrict the abstract syntax to precisely the subset of Java representable in our Coq formalisation.

Therefore, the entire Java code was first parsed to an abstract syntax tree, which we tried to transform to our precise AST by introducing temporary variables. If the transformation failed, we asked the user to simplify the original code. It is crucial that the structure of the original Java code and the transformed code stays structurally equivalent, because the user should recognize the code when reasoning about it.

However, the custom parser turned out to have a high impact on the time that a user must wait for the first time she attempts to verify

Gen	Active Until	Java Parser	Proof Integration	Coq Interaction	Kloc	Coq version	Eclipse version
0.0.*	Jun '12	custom	SM	stdout	7.9	8.2	3.6
0.1.*	Oct '12	custom	AOP	stdout	9.5	8.2-8.3	3.6-3.7
0.2.*	Mar '13	Eclipse	AOP	stdout	5.5	8.2-8.4	3.6-3.7
0.3.*	active	Eclipse	AOP	XML	4.8	$\geq$8.4	$\geq$4.2
0.4.*	future	Eclipse	EMF	XML	n/a	$\geq$8.4	$\geq$4.2

Table 8.1: Comparison of features of different Kopitiam generations.

a method. The Scala compiler generated over 1100 classes from the combinator-based parser, which involves many anonymous functions, translated to classes. The JVM class loader is not designed to load such a large number of classes in a short time.

Since the third generation Kopitiam uses the Eclipse Java AST, which we use to check whether an AST is understandable by our Coq formalisation. We also use the Eclipse Java AST to output the Coq abstract syntax representing the Java program. This code is much faster since the AST is already parsed by Eclipse, and all classes are loaded upfront by Eclipse.

8.6 Conclusion

Table 8.1 summarizes the main features of all Kopitiam generations, and also shows the planned future generation. The first column reports the generation and the second until when it was actively developed. The third column describes the Java parser we are using: either our custom combinator parser (custom) or the parser provided by Eclipse (Eclipse). The fourth column presents the proof script and specification integration, either calls to static methods (SM), antiquotes by using aspect-oriented programming to change Eclipse's built-in lexer and parser (AOP), or the Eclipse modeling framework (EMF), which is planned for the future. The fifth column describes the Coq interaction, either screen-scraping from output streams (stdout) or the structured XML interface, only available since Coq 8.4. The sixth column provides the number of source lines of Kopitiam, in thousands of lines. We used `wc` to count the lines of the latest release of each generation.

The supported Coq versions and Eclipse versions are shown in the last two columns.

We can clearly see that eliminating our custom combinator-parser reduced the code size, as did adopting the structured XML interface of Coq.

Kopitiam is still a work in progress, but we now have a much better idea what we want to achieve and how to get there. Kopitiam will continue to adapt to new features of both Coq and Eclipse, and workarounds that had to be developed due to limitations of these external software will be removed at a later point.

The core functionality is robust and in place now, which serves as a solid basis for more advanced features. Evidence for this statement is that Kopitiam survived with only minor adjustments the updating of major releases of Scala, Coq and Eclipse. The latest generation of Kopitiam (0.3.*) has a significantly reduced amount of global state and is now using one `coqtop` process for each editor. We schedule the sending of commands to the Coq processes in background threads so that the user interface is responsive.

We evaluated the 0.2.* generation by using it for teaching a master's-level course. This course was based on Pierce et al.'s *Software Foundations* [99] book, which we taught in the first 8 weeks. We asked the participants to fill out a survey about Kopitiam, the results of which are very promising.

8.7 Future Work

We briefly describe the future directions of the development of Kopitiam. Kopitiam can be thought of as containing two main separate components, the Coq development mode and the extension to the Java development environment.

The Coq development mode should be extended with common features from development environments for other programming languages:

- We want to support *dependency tracking*: whenever the user modifies the Coq development, automated builder jobs should be run in the background to recompile the dependent files.

- We will also look into *navigation* within Coq projects. A user should be able to jump to the definition of a lemma or data type.
- *Refactoring* of Coq code, like renaming a definition, extraction of helper lemmas from within a proof.
- To accomplish being a complete Coq environment, we want to deal with Coq's *notation* feature.

In the Java extension, we want to include features to produce useful documentation from a verified program, and also to simplify proofs by providing better tools to the user.

- Verifyied programs should produce *documentation* which states the proven correctness lemmas integrated into Javadoc output.
- We want to develop a concise *visualization* of the symbolic heap and stack, instead of providing only the textual output of Coq's proof obligations. This will improve the user experience.
- *Context-sensitive suggestions* for developing specifications and proof script should be provided, like completions of local variables and field names.
- The *specification syntax* should be improved. We will likely switch to using a modeling framework like EMFText[3] for our customized syntax.
- We will possibly adopt an already existing *specification language* like JML. This would allow us to apply more case studies based on the case studies done with JML.
- Integration into the Java program *refactoring*: for example, when a field is renamed, the specifications refering to this field should be updated automatically.
- Implement *local transformations* of Java programs, such as nested field access, to support more Java programs.

[3] `http://www.emftext.org/`

- Kopitiam supports only a subset of Java, which should be extended with more features like *generics* and *arrays*. This requires extensions to Charge! as well.

We have presented a roadmap of possible future features of Kopitiam. At the moment a full-time programmer is working on Kopitiam, along with a MSc student, who focuses on the specification language integration.

Chapter 9

Empirical Evaluation of Kopitiam

Abstract

In this paper we present an empirical evaluation of Kopitiam based on the technology acceptance model. Kopitiam is an Eclipse plugin that integrates the proof assistant Coq with development of Java code. After we taught a group of M.Sc. students Coq, we asked them to fill out a questionnaire, whose results are presented in this empirical evaluation.

9.1 Introduction

Kopitiam is an Eclipse plugin to integrate proof development in Coq with program development in Java. In this paper we focus solely on the Coq development environment, rather than the Java integration. We taught a one semester (16 weeks) course at the master's level based on material from Pierce et al's book *Software Foundations* [99]. To evaluate Kopitiam's usefulness, we conducted an empirical evaluation of the current version (at the beginning of February 2013) of Kopitiam, which was the 0.2.* generation. The 0.2.* generation of Kopitiam supports Coq 8.2 and above, and uses screen scraping of the standard output to communicate with Coq. A detailed comparison between the Kopitiam generations is given in Chapter 8. We analyze the data collected at the end of the semester in this paper. The research framework chosen for this analysis is Davis's technology acceptance model [40] from 1989, which is a theory that models how users come to accept and use a technology.

In this paper, we will present our research objective in Section 9.2, the methodology we used in Section 9.3, our concrete questionnaire in Section 9.4, the participants in Section 9.5, the results of the questionnaire in Section 9.6, threats to validity in Section 9.7. In Section 9.8 we discuss the implications of these results and we conclude the paper in Section 9.9.

9.2 Research Objective

Our primary goal in this study was to investigate the usability of Kopitiam by students at the master's level. To achieve this goal we designed a pilot evaluation investigating the perceived usefulness, perceived ease of use, and self-predicted future use by the study participants. Furthermore, we are interested whether the participants propose to use Kopitiam in the future for teaching, ways to make it easier to get started with Kopitiam, Kopitiam's robustness, and desired features. To address these matters, we used a questionnaire based on an adapted version of the technology acceptance model [71]. The key research questions that were addressed by our study are:

- RQ1: Do users perceive Kopitiam as useful, not useful or are they indifferent?
- RQ2: Do users perceive Kopitiam as easy to use, difficult to use or are they indifferent?
- RQ3: Would users adopt Kopitiam in the future?
- RQ4: Do users of Kopitiam suggest that we continue using Kopitiam in future courses?
- RQ5: What is the main source of confusion, Kopitiam, Coq or Eclipse?
- RQ6: What would make it easier to get started with Kopitiam?
- RQ7: Is Kopitiam robust?
- RQ8: How does Kopitiam compare to other verification tools?
- RQ9: What are desired features of Kopitiam?

9.3 Methodology of the Evaluation

The technology acceptance model [40] aims at assessing user beliefs about the usefulness and ease of use of a technology by using a questionnaire. The questionnaire focusses on both perceived usefulness and perceived ease of use. Davis [40] defines the term "usefulness" as "the degree to which a person believes that using a particular system would enhance his or her job performance" and the term "perceived ease of use" as "the degree to which a person believes that using a particular system would be free of effort".

In this study we use an extended version of the technology acceptance model, which was proposed by Laitenberger and Dreyer [71], and also evaluates self-predicted future use.

The first three variables (RQ1, RQ2, and RQ3) were measured with a seven-point Likert scale, which allows positive, negative, and neutral evaluation to be captured. The Likert scale ranges from 1 to 7, where 1 corresponds to "extremely likely", 2 to "quite likely", 3 to "slightly likely", 4 to "neither", 5 to "slightly unlikely", 6 to "quite unlikely" and 7 to "extremely unlikely".

The proposed usage of Kopitiam (RQ4) variable was measured with two possible answers: yes and no. This methodology allows only positive and negative evaluation to be captured.

The source of confusion variable (RQ5) was measured with three possible answers: Coq, Eclipse, and Kopitiam. This methodolgy assigns confusion to a single component.

The remaining variables (RQ6 – RQ9) in the questionnaire were measured with open-ended questions (text fields): what would make Kopitiam easier to get started, the robustness of Kopitiam, how Kopitiam compares to other verification tools, and what the desired features of Kopitiam are.

9.4 Questionnaire

In the following we present the questionnaire used for each variable, where the first two are taken from the technology acceptance model [40], and the third from the extension for self-predicted future use [71].

Perceived Usefulness

- U1: Using Kopitiam in my job would enable me to accomplish tasks more quickly.
- U2: Using Kopitiam would improve my job performance.
- U3: Using Kopitiam in my job would increase my productivity.
- U4: Using Kopitiam would enhance my effectiveness on the job.
- U5: Using Kopitiam would make it easier to do my job.
- U6: I would find Kopitiam useful in my job.

Perceived Ease of Use

- E1: Learning to operate Kopitiam would be easy for me.
- E2: I would find it easy to get Kopitiam to do what I want it to do.
- E3: My interaction with Kopitiam would be clear and understandable.
- E4: I would find Kopitiam to be flexible to interact with.
- E5: It would be easy for me to become skillful at using Kopitiam.
- E6: I would find Kopitiam easy to use.

Self-predicted Future Use

- F1: Assuming Kopitiam would be available on my job, I predict that I will use it on a regular basis in the future.
- F2: I would prefer using Kopitiam to paper-based techniques for performing proofs.

Proposed Future Use (answer possibilities: "Yes" or "No")

- PF1: Would you suggest to use Kopitiam in future courses?
- PF2: Would you expect your fellow students to be able to use Kopitiam?

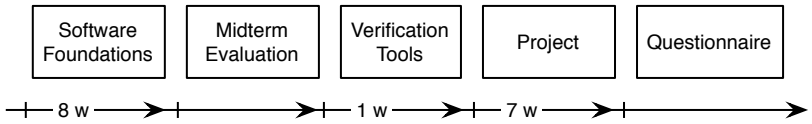


Figure 9.1: Timeline of the semester: the first 8 weeks had "Software Foundations" as their topic, followed by the midterm evaluation. Afterwards one week was spent on different verification tools, which was followed by a 7 week project. At the end we gathered feedback with the questionnaire.

Source of Confusion (answer possibilities: "Coq", "Eclipse" or "Kopitiam")

- C1: What is the main source of confusion while proving?

Robustness and Suggestions (open-ended questions)

- G1: What would it make easier to get started?
- G2: Did Kopitiam crash in your experience? If so, were you able to recover from the crash or got support?
- G3: Have you ever been lost while using Kopitiam? If so, please recall the circumstances. How did you continue?
- G4: What are possible future extensions for Kopitiam? In which area should Kopitiam improve the most?
- G5: We briefly looked at other verification tools. Please elaborate what in your opionion the main difference between Kopitiam and other verification tools is.
- G6: General feedback

9.5 Participants and Setup

We recruited participants from the master's course that we taught during the Spring Semester 2013 (16 weeks, 2 lectures with 2 hours each week). The master's course is mandatory for students of the programming languages specialization in ITU's software engineering master's program. A student can choose between several different specializations

in the software engineering program. The background of a student starting a master in software engineering at ITU is a completed B.Sc. degree in any subject. In the course in spring 2013 a zoologist and an architect participated, together with several software engineers. Some had some functional programming background, the majority did not. None of the students had done formal proofs before, some have been taught logical foundations before. The timeline of the semester is shown in Figure 9.1. We started teaching Pierce et al's *Software Foundations* [99] during the first eight weeks. The usual midterm course evaluation was just after the eighth week. In the ninth week, the students were asked to present other verification tools in groups. They chose VeriFast [58], Spec# [7] and the dependently typed programming language Idris [27]. In the last seven weeks, the students had to work in groups on a project where they implement and verify some data structure or extend an existing formalization. One group chose to use VeriFast for the project, while the remaining four groups used Coq for their projects.

The students were asked to use Kopitiam for all mandatory exercises and the project. However, some students decided to use CoqIDE instead of Kopitiam.

At the end of the course, all students were asked to fill out the questionnaire, which was available online. The answers were recorded anonymously. Six out of thirteen students who participated in the course filled out the questionnaire. All recorded answers were from students who used Kopitiam, rather than CoqIDE.

In order to preserve anonymity with a small respondent group we did not record any demographic information for the questionnaire.

9.6 Results

We present the aggregate results of the self-reported quantitative data in this section, and evaluate the answers to the open-ended questions in the discussion (Section 9.8). In general, Kopitiam received good results from the evaluation.

Perceived Usefulness is presented in Table 9.1. Kopitiam is perceived to be appropriate for performing tasks more quickly (U1) and more easily (U5). The participants are less convinced, but still between "slightly"

	$\overline{x}$	$\tilde{x}$	σ
U1	2.17	2.00	0.37
U2	2.50	2.50	0.50
U3	2.50	2.50	0.50
U4	2.67	3.00	0.47
U5	2.33	2.00	0.47
U6	2.50	2.50	0.50

Table 9.1: Perceived usefulness of Kopitiam, for each question U1-U6, the table shows the arithmetic mean ($\overline{x}$), the statistical median ($\tilde{x}$) and the standard deviation (σ). The data was gathered with a seven point Likert scale ranging from 1 (extremely likely) to 7 (extremely unlikely).

	$\overline{x}$	$\tilde{x}$	σ
E1	1.83	2.00	0.69
E2	2.00	2.00	0.58
E3	2.00	2.00	0.58
E4	3.17	3.00	0.69
E5	2.17	2.00	0.69
E6	2.17	2.00	0.69

Table 9.2: Perceived ease of use of Kopitiam, for each question E1-E6, the table shows the arithmetic mean ($\overline{x}$), the statistical median ($\tilde{x}$) and the standard deviation (σ). The data was gathered with a seven point Likert scale ranging from 1 (extremely likely) to 7 (extremely unlikely).

and "quite" likely, of the improved performance (U2), productivity (U3), and general usefulness (U6). The participants are tending towards "quite likely" for the effectiveness (U4) enhancements.

Perceived Ease of Use is presented in Table 9.2. The participants found Kopitiam to be extremely easy to learn (E1). They perceived the navigation as quite easy (E2), and the user interface as clear (E3). Their experience was that they became quite skilled (E5) and their overall impression is that Kopitiam is quite easy to use (E6). The participants were not completely positive regarding the flexibility of Kopitiam (E4).

Self-Predicted Future Use is presented in Table 9.3. The participants extremely prefer the verification tool Kopitiam over paper-based proofs (F2), whereas they are not too sure that they will use Kopitiam on a regular basis (F1).

	$\overline{x}$	$\tilde{x}$	σ
F1	2.50	2.00	0.50
F2	1.50	2.50	0.76

Table 9.3: Self-predicted future use of Kopitiam, for both questions F1 and F2, the table shows the arithmetic mean ($\overline{x}$), the statistical median ($\tilde{x}$) and the standard deviation (σ). The data was gathered with a seven point Likert scale ranging from 1 (extremely likely) to 7 (extremely unlikely).

Proposed Future Use The unison answer of all participants to the proposed future use questions (PF1 and PF2) was "Yes".

Source of Confusion 5 out of the 6 participants (83%) found Coq to be the main source of confusion (C1), whereas a single participant found Eclipse the main source of confusion.

Robustness and Suggestions The results of the open-ended questions are evaluated in Section 9.8.

9.7 Threats to Validity

Participants' Affiliation All participants were master's students from IT University of Copenhagen who were enrolled in our course. This potentially imposes a bias, but the participants chose voluntarily to answer the questionnaire. It was clearly communicated to the participants that the results would be recorded anonymously and it was not mandatory to fill out the questionnaire.

Participants' Training The participants were trained in the usage of Kopitiam during one semester (16 weeks). They were not trained in other verification tools, and they have not seen an interactive proof assistant before. The comparison to other verification tools is shallow, since these were only briefly presented.

Participants' Partiality The participants were taught by us for a semester, and are graded by us, which might impose a bias. Filling out the questionnaire was not mandatory, and we recorded the answers anonymously. Roughly half of the course participants filled out the questionnaire, which shows that it was completely voluntary.

	α
Usefulness	0.87
Ease of use	0.91
Future use	0.83

Table 9.4: Cronbach's α of the investigated variables.

Validity of Results This is a pilot experiment; thus, the number of participants was limited. The results are not generalizable or scalable due to the small participating group. We analyzed the reliability of the evaluation by looking into the internal consistency of the three investigated variables of the technology acceptance model [40]. To accomplish that, we computed Cronbach's alpha coefficient [37], by using the methodology presented by Bland and Altman [22]. Our results are shown in Table 9.4. The threshold for acceptable results is 0.7, as presented by Bland and Altman [22]. In our case, the perceived usefulness and self-predicted future use show good internal consistency (value is greater than 0.8). The perceived ease of use has an extremely good internal consistency (value greater than 0.9). The results of the experiment are not generalizable, but the α gives good confidence in the internal consistency of the gathered results.

9.8 Discussion

The purpose of Kopitiam is to provide a development environment for Coq proofs. We presented the aggregated results of the quantitative data using the technology acceptance model in Section 9.6. Those results are promising for the future use and development of Kopitiam.

Before our questionnaire at the end of the semester, a standard course evaluation was conducted in the middle of the semester, as shown in Figure 9.1. Already in this course evaluation, motivating comments about Kopitiam were present: "The Kopitiam plugin [...] works very well" and "Kopitiam makes Coq easier to comprehend".

The answers to how to get easier started with Kopitiam (G1) included "better keyboard shortcuts" and "better compatibility with different Eclipse versions". The keyboard shortcuts are configurable by a user, but we failed to mention this. Kopitiam supports different Eclipse

versions, but we were too conservative and had a dependency that was too tight on a specific Eclipse version. This tight dependency was loosened in the fourth generation.

The answers to whether Kopitiam crashed (G2) varied a lot, from "It did not crash, but I had to restart Coq plenty of times." to "Yes, Kopitiam did crash. [...] the crashes happened more frequently the more lines of code I put into a single file".

During the semester we fixed only one issue in Kopitiam: the parsing of comments lead to a stack overflow on Windows. The last comment indicates that an old release without the fix for the issue was used, and no upgrade was performed by the student.

One participant replied extensively by observing that "Kopitiam [...] stalled [...] in the following cases:"

- "The .v [source] file was changed outside of Eclipse".
- "The .vo [object] file was changed while it was loaded and you continued [to interact]".
- "If you used 'step to cursor' on a point that was far behind the current cursor".
- "Sometimes, stepping forward over trivial tactics would take large amount of time [...]".

In a new Kopitiam generation the communication interface with Coq is rewritten, and uses the newly available structured input and output of Coq. This structured communication interface results in fewer crashes and more reliable error reporting from Coq.

There were only two affirmative answers to whether the participant felt at any time lost at any time (G3). One mentioned the issue of the communication protocol with Coq, which has been solved in the fourth generation "If your assumption-list grows to large, it overflows the box, and spills into the console. [...] I continued by using another approach that did not produce such large assumption-lists.". The other answer was "Yes, but an update solved the problem.".

The proposed future extensions (G4) ranged over "Just eliminate the bugs and improve the hotkeys." and "When selecting different goals, it would be a huge improvement if the context changed as well.". Another

proposal was "[...] better syntax highlighting [...], code completion for tactics, adding external Coq libraries [...] from within the Eclipse environment". "An integrated build system [...]" was mentioned several times, we are working on this.

The comparison to other verification tools (G5) is biased due to their extensive training with Kopitiam, and only a shallow introduction into other verification tools. The answers were "Compared to a tool like VeriFast, Kopitiam makes it a lot clearer what proof obligations you have and what you have in the context" and "Compared to the way you currently do verification and implementation in Idris, the interface of Kopitiam seems once again to provide a much cleaner presentation of proof obligations and context than the command-line tools of Idris". One answer compared Kopitiam to other courses: "It is more thorough but also a bit more theoretical than practical with regards to 'regular' software development.".

The general feedback (G6) is promising for future development: "Quite a decent plugin. Not a fan of Eclipse in general, but Kopitiam worked quite well, and was very easy to use." "[...] All in all, I think Kopitiam is a great tool to work with, although it does crash a bit too often."

9.9 Conclusion

We continue the development of Kopitiam, and are eager to evaluate the next generation of Kopitiam, after addressing the issues identified. It would be very interesting to have a comparative study of Kopitiam and other verification tools, but it is not easily possible to teach multiple verification tools in depth in a single semester. We teach the students Kopitiam and Coq in one semester, and teaching them another verification tool, being it a different proof system or another Coq development environment (e.g. Proof General or CoqIDE), would take much more time. It is crucial to understand the differences between the verification a user can do in different tools and the subtle differences between these tools, and this cannot be taught within a small period of time.

Part III

Research Papers: Case Studies

Chapter 10

Formalized Verification of Snapshotable Trees: Separation and Sharing

Originally published in: VSTTE 2012 [82]

Joint work with: Filip Sieczkowski, Lars Birkedal, and Peter Sestoft; ITU Copenhagen

Abstract

We use separation logic to specify and verify a Java program that implements snapshotable search trees, fully formalizing the specification and verification in the Coq proof assistant. We achieve *local* and *modular* reasoning about a tree and its snapshots and their iterators, although the implementation involves shared mutable heap data structures with no separation or ownership relation between the various data.

The paper also introduces a series of four increasingly sophisticated implementations and verifies the first one. The others are included as future work and as a set of challenge problems for full functional specification and verification, whether by separation logic or by other formalisms.

10.1 Introduction

This paper presents a family of realistic but compact challenge case studies for modular software verification. We fully specified and verified the first case study in Coq, using a domain-specific separation logic [68] and building upon our higher-order separation logic [12]. As future work we

plan to verify the other implementations with the presented abstract interface specification. We believe this is the first mechanical formalization of this approach to modular reasoning about implementations that use shared heap data with no separation or ownership relation between the various data.

The family of case studies consists of a single interface specification for snapshotable trees, and four different implementations. A *snapshotable tree* is an ordered binary tree that represents a set of items and supports taking readonly *snapshots* of the set, in constant time, at the expense of slightly slower subsequent updates to the tree. A snapshotable tree also supports iteration (enumeration) over its items as do, e.g., the Java collection classes. The four implementations of the snapshotable tree interface all involve shared heap data as well as increasingly subtle uses of destructive heap update.

For practical purposes it is important that the same interface specification can support verification of multiple implementations with varying degrees of internal sharing and destructive update. Moreover, the specification must accommodate any number of data structure (tree) instances, each having any number of iterators and snapshots, each of which in turn can have any number of iterators. Most importantly, we show how we can have local reasoning (a frame rule) even though the tree and its snapshots share mutable heap data.

We welcome other solutions to the specification and verification of this case study; indeed R. Leino has already made one (unpublished) using Dafny [74].

The Java source code of the case studies of all four implementations and the Coq source is available at `http://www.itu.dk/people/hame/snapshots.tgz`.

Section 10.2 presents the interface of the case study data structure, shows an example use, and outlines four implementations. Section 10.3 gives a formal specification of the interface using separation logic and verifies the example code. Sections 10.4 and 10.5 verify the first implementation.

10.2 Case Study: Snapshotable Trees

The case study is a simplified version of snapshotable treesets from the C5 collection library [66].

10.2.1 Interface: Operations on Snapshotable Trees

Conceptually, a snapshot of a treeset is a readonly copy of the treeset. Subsequent updates to the tree do not affect any of its snapshots, so one can update the tree while iterating over a snapshot. Taking a snapshot must be a constant time operation, but subsequent updates to the tree may be slower after a snapshot has been taken. Implementations (Section 10.2.3) typically achieve this by making the tree and its snapshots share parts of their representation, gradually unsharing it as the tree gets updated, in a manner somewhat analogous to copy-on-write memory management schemes in operating systems.

All tree and snapshot implementations implement the same ITree interface:

```
public interface ITree extends Iterable<Integer> {
  public boolean contains(int x);
  public boolean add(int x);
  public ITree snapshot();
  public Iterator<Integer> iterator();
}
```

These operations have the following effect:

- `tree.contains(x)` returns true if the item is in the tree, otherwise false.
- `tree.add(x)` adds the item to the tree and returns true if the item was not already in the tree; otherwise does nothing and returns false.
- `tree.snapshot()` returns a readonly snapshot of the given tree. Updates to the given tree will not affect the snapshot. A snapshot cannot be made from a snapshot.

- `tree.iterator()` returns an iterator (also called enumerator, or stream) of the tree's items. Any number of iterators on a tree or snapshot may exist at the same time. Modifying a tree will invalidate all iterators on that tree (but not on its snapshots), so that the next operation on such an iterator will throw a ConcurrentModificationException.

We include the somewhat complicated `iterator()` operation because it makes the distinction between a tree and its snapshots completely clear: While it is illegal to modify a tree while iterating over it, it is perfectly legal to modify the tree while iterating over one of its snapshots. Also, this poses an additional verification challenge when considering implementations with rebalancing (cases A2B1 and A2B2 in Section 10.2.3) because `tree.add(item)` may rebalance the tree in the middle of an iteration over a snapshot of the tree, and that should be legal and not affect the iteration.

Note that for simplicity, items are here taken to be integers; using techniques from [110] it is straightforward to extend our formal specification and verification to handle a generic version of snapshotable trees.

10.2.2 Example Client Code

To show what can be done with snapshots and iterators (and not without), consider this piece of client code. It creates a treeset `t`, adds three items to it, creates a snapshot `s` of the tree, and then iterates over the snapshot's three items while adding new items (6 and 9) to the tree:

```
ITree t = new Tree();
t.add(2); t.add(1); t.add(3);
ITree s = t.snapshot();
Iterator<Integer> it = s.iterator();
boolean lc = it.hasNext();
while (lc) {
  int x = it.next();
  t.add(x * 3);
  lc = it.hasNext();
}
```

10.2.3 Implementations of Snapshotable Trees

One may consider four implementations of treesets, spanned by two orthogonal implementation features. First, the tree may be unbalanced (A1) or it may be actively rebalanced (A2) to keep depth $O(\log n)$. Second, snapshots may be kept persistent, that is, unaffected by tree updates, either by path copy persistence (B1) or by node copy persistence (B2):

	Without rebalancing	With rebalancing
Path copy persistence	A1B1	A2B1
Node copy persistence	A1B2	A2B2

The implementation closest to that of the C5 library [66, section 13.10] is A2B2, which is still somewhat simplified: only integer items, no comparer argument, no update events, and so on. In this paper we formalize and verify only implementation A1B1; the verification of the more sophisticated implementations A1B2, A2B1 and A2B2 will be addressed in future work.

Nevertheless, for completeness and in the hope that others may consider this verification challenge, we briefly discuss all four implementations and the expected verification challenges here.

With *path copy persistence* (cases AxB1), adding an item to a tree will duplicate the path from the root to the added node, if this is necessary to avoid modifying any snapshot of the tree. Thus an update will create $O(d)$ new nodes where d is the depth of the tree.

With *node copy persistence* (cases AxB2), each tree node has a spare child reference. The first update to a node uses this spare reference, does not copy the node and does not update its parent; the node remains shared between the tree and its snapshots. Only the second update to a node copies it and updates its parent. Thus an update does not replicate the entire path to the tree root; the number of new nodes per update is amortized $O(1)$. See Driscoll [46] or [66].

To implement ordered trees without rebalancing (cases A1By), we use a Node class containing an item (here an integer) and left and right children; `null` is used to indicate the absence of a child. A tree or snapshot contains a stamp (indicating the "time" of the most recent update) and a reference to the root Node object; `null` if the tree is empty.

To implement rebalancing of trees (cases A2By), we use left-leaning red-black trees (LLRB) which encode 2-3 trees [2, 105], instead of general red-black trees [53] as in the C5 library. This reduces the number of rebalancing cases.

To implement iterators on a tree or snapshot we use a class TreeIterator that holds a reference to the underlying tree, a stamp (the creation "time" of the iterator) and a stack of nodes. The stamp is used to detect subsequent updates to the underlying tree, which will invalidate the iterator. Since snapshots cannot be updated, their iterators are never invalidated. The iterator's stack holds its current state: for each node in the stack, the node's own item and all items in the right subtree have yet to be output by the iterator.

Case A1B1 = no rebalancing, path copy persistence In this implementation there is shared data between a tree and its snapshots, but the shared data is not being mutated because the entire path from the root to an added node gets replicated. Hence no node reachable from the root of a snapshot, or from nodes in its iterators' stacks, can be affected by an update to the live tree; therefore no operation on a snapshot can be affected by operations on the live tree. Although this case is therefore the simplest case, it already contains many challenges in finding a suitable specification for trees, snapshots and iterators, and in proving the stack-based iterator implementation correct.

Case A2B1 = rebalancing, path copy persistence In this case there is potential mutation of shared data, because the rebalancing rotations seem to be able to affect nodes just off the fresh unshared path from a newly added node to the root. This could adversely affect an iterator of a snapshot because a reference from the iterator's node stack might have its right child updated (by a rotation), thus wrongly outputting the items of its right subtree twice or not at all. However, this does not happen because the receiver of a rotation (to be moved down) is always a fresh node (we're in case B1 = path copy persistence) and moreover we consider only `add` operations (not `remove`), so the child being rotated (moved up) is also a fresh node and thus not on the stack of any iterator – the rebalancing was caused by this child being "too deep" in the tree. Hence if we were to support `remove` as well, then perhaps the implementation of rotations needs to be refined.

Case A1B2 = no rebalancing, node copy persistence In this case, there is mutation of shared data not observable by the client. For example, a left-child update to a tree node that is also part of a snapshot will move the snapshot's left-child value to the node's extra reference field, and destructively update the left child as required for the live tree. There should be no observable change to the snapshot, despite the change to the data representing it. The basic reason for correctness is that any snapshot involving an updated node will use the extra reference and hence not see the update; this is true for nodes reachable from the root of a snapshot as well as for nodes reachable from the stack of an iterator. When we need to update a node whose extra reference is already in use, we leave the old node alone and create a fresh copy of the node for use in the live tree; again, existing snapshots and their iterators do not see the update.

Case A2B2 = rebalancing, node copy persistence In this case there is mutation of shared data (due both to moving child fields to the extra reference in nodes, and due to rotations), not observable for the client. Since the updates caused by rotations are handled exactly like other updates, the correctness of rebalancing with respect to iterators seems to be more straightforward than in case A2B1.

10.3 Abstract Specification and Client Code Verification

We use higher-order separation logic [100, 20] to specify and verify the snapshotable tree data structure. We build on top of our intuitionistic formalization of HOSL in Coq [12] with semantics for an untyped Java-like language.

To allow implementations to share data between a tree, its snapshots, and iterators and still make it possible for clients to reason locally (to focus only on a single tree / snapshot / iterator), we will use an idea from [68] (see also the verification of Union-Find in [67]). The idea is to introduce an abstract predicate, here named H, global to each tree data structure consisting of a single tree, multiple snapshots, and multiple iterators. This abstract predicate H is parameterized by a finite set of disjoint *abstract structures*. We have three kinds of abstract structures: Tree, Snap, and Iter. The use of H enables a client of our specification to

consider each abstract structure to be separate or disjoint from the rest of the abstract structures and thus the client can reason modularly about client code using only those abstract structures she needs; the rest can be framed out. Since the abstract predicate H is existentially quantified, the client has no knowledge of how an implementation defines H (see [20, 94] for more on abstract predicates in higher-order separation logic). The implementor of the tree data structure has a global view on the tree with its snapshots and iterators, and is able to define which parts of the abstract structures are shared in the concrete heap. Section 10.4 defines H for the A1B1 case from Section 10.2.3.

The Tree abstract structure consists of a handle (reference) to the tree and a model, which is an ordered finite set, containing the elements of the tree. The Snap structure is similar to Tree. The Iter structure consists of a handle to the iterator and a model, which is a list containing the remaining elements for iteration. Because H is tree-global, exactly one Tree structure must be present ("the tree"), while the number of Snap and Iter structures is not constrained.

10.3.1 Specification of the ITree Interface

We now present the formal abstract specification of the ITree interface informally described in Section 10.2.1. The specification also contains five axioms, which are useful for a client and obligations to an implementor of the interface. The specification is parametrized over an implementation class C and the above-mentioned predicate H, and each method specification is universally quantified over the model τ, a finite set of integers and a finite set of abstract structures ϕ.

interface ITree {

$H(\{\mathit{Tree}(\texttt{this},\tau)\} \uplus \phi)$	`contains(x)`	$\texttt{ret} = \texttt{x} \in \tau \wedge$ $H(\{\mathit{Tree}(\texttt{this},\tau)\} \uplus \phi)$
$H(\{\mathit{Snap}(\texttt{this},\tau)\} \uplus \phi)$	`contains(x)`	$\texttt{ret} = \texttt{x} \in \tau \wedge$ $H(\{\mathit{Snap}(\texttt{this},\tau)\} \uplus \phi)$
$H(\{\mathit{Tree}(\texttt{this},\tau)\} \uplus \phi)$	`add(x)`	$\texttt{ret} = \texttt{x} \notin \tau \wedge$ $H(\{\mathit{Tree}(\texttt{this},\{\texttt{x}\} \cup \tau)\} \uplus \phi)$
$H(\{\mathit{Tree}(\texttt{this},\tau)\} \uplus \phi)$	`snapshot()`	$H(\{\mathit{Snap}(\texttt{ret},\tau), \mathit{Tree}(\texttt{this},\tau)\} \uplus \phi)$
$H(\{\mathit{Snap}(\texttt{this},\tau)\} \uplus \phi)$	`iterator()`	$H(\{\mathit{Iter}(\texttt{ret},[\tau]), \mathit{Snap}(\texttt{this},\tau)\} \uplus \phi) \wedge$ $\texttt{ret} <: \mathit{Iterator}$

(a) $H(\{\mathit{Tree}(\texttt{t},\tau)\} \uplus \phi) \vdash \texttt{t} : C$
(b) $H(\{\mathit{Snap}(\texttt{s},\tau)\} \uplus \phi) \vdash \texttt{s} : C$
(c) $\tau = \tau' \wedge H(\{\mathit{Tree}(\texttt{t},\tau)\} \uplus \phi) \vdash H(\{\mathit{Tree}(\texttt{t},\tau')\} \uplus \phi)$
(d) $H(\{\mathit{Snap}(\texttt{s},\tau)\} \uplus \phi) \vdash H(\phi)$
(e) $H(\{\mathit{Iter}(\texttt{it},\alpha)\} \uplus \phi) \vdash H(\phi)$

}

These specifications can be read as follows:

- `contains` requires either a Snap or Tree structure (written as separate specifications) for the `this` handle and some set τ. The structure is unmodified in the postcondition, and the return value `ret` is true if the item `x` is in the set τ, otherwise false.
- `add` requires a Tree structure for the `this` handle and some set τ. The postcondition states that the given item `x` is added to the set τ. The return value indicates whether the tree was modified, which is the case if the item was not already present in the set τ.
- `snapshot` requires a Tree structure for the `this` handle and some set τ. The postcondition constructs a Snap structure for the returned handle `ret` and the set τ. So the Tree and the Snap structure contain the same elements.
- `iterator` requires a Snap structure for the `this` handle and some set τ. The postcondition constructs an Iter structure with the return handle and the set τ converted to an ordered list, written $[\tau]$.

The returned handle conforms (written $<:$) to the Iterator specification shown in Section 10.3.2.

The five axioms state that (a) the static type of the tree is the given class C; (b) the static type of a snapshot is C; (c) the model τ of the tree can be replaced by an equal model τ' [1]; and we can forget about snapshots (d) and iterators (e).

In contrast to the description in Section 10.2.1 we leave iterators over the tree for future work. We could use the ramification operator [68] to express that any iterators over the tree become invalid when the tree is modified.

The abstract separation can be observed, e.g., in the specification of `add`: it only modifies the model of the Tree structure and does not affect the rest of the abstract structures (ϕ is preserved in the postcondition). Hence the client can reason about calls to `add` locally, independently of how many snapshots and iterators there are.

In our Coq formalization we do not have any syntax for interfaces at the specification logic level [12], but represent interfaces using Coq-level definitions. Appendix 10.8 contains the formal representations (ITree, Iterator, Stack).

10.3.2 Iterator Specification

Our iterator specification is also parametrized over a class IC and a predicate H, and each method specification is universally quantified over a list of integers α and a finite set of abstract structures ϕ.

interface `Iterator<Integer>` {

$H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)$	`hasNext()`	$\texttt{ret} = (\lvert\alpha\rvert \neq 0) \wedge H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)$
$H(\{Iter(\texttt{this}, x :: \alpha)\} \uplus \phi)$	`next()`	$\texttt{ret} = x \wedge H(\{Iter(\texttt{this}, \alpha)\} \uplus \phi)$

}

The specification of the Iterator interface requires an Iter structure with the `this` handle and some list α. The return value of the method `hasNext` captures whether the list α is non-empty. The Iter structure in the postcondition is not modified. The method `next` requires an Iter

[1] This is explicit for technical reasons: in our implementation H is defined inside a monad [12], and the client should not have to discharge obligations inside the monad.

```
1: t.add(2);t.add(1);t.add(3);
```
$\{H(\{Tree(\texttt{t},\{1,2,3\})\})\}$
```
2: ITree s = t.snapshot();
```
$\{H(\{Snap(\texttt{s},\{1,2,3\}),Tree(\texttt{t},\{1,2,3\})\})\}$
```
3: Iterator<Integer> it = s.iterator();}
```
$\{H(\{Iter(\texttt{it},[1,2,3]),Tree(\texttt{t},\{1,2,3\}),Snap(\texttt{s},\{1,2,3\})\})\}$
```
4: boolean lc = it.hasNext();
```
$\{\texttt{lc} = \texttt{true} \wedge H(\{Iter(\texttt{it},[1,2,3]),Tree(\texttt{t},\{1,2,3\}),Snap(\texttt{s},\{1,2,3\})\})\}$
```
5: while (lc) \{
  invariant:
```
$\exists\alpha,\beta.\alpha@\beta = [1,2,3] \wedge \texttt{lc} = (|\beta| \neq 0) \wedge$
$H(\{Tree(\texttt{t},\{1,2,3\} \cup \{3z|z \in \alpha\}),Snap(\texttt{s},\{1,2,3\}),Iter(\texttt{it},\beta)\})$
```
6:   int x = it.next();
```
$\{\alpha@\beta = [1,2,3] \wedge \texttt{lc} = (|\beta| \neq 0) \wedge \beta = \texttt{x} :: \beta' \wedge$
$H(\{Tree(\texttt{t},\{1,2,3\} \cup \{3z|z \in \alpha\})Snap(\texttt{s},\{1,2,3\}),Iter(\texttt{it},\beta')\})\}$
```
7:   t.add(x * 3);
```
$\{\alpha@\beta = [1,2,3] \wedge \texttt{lc} = (|\beta| \neq 0) \wedge \beta = \texttt{x} :: \beta' \wedge$
$H(\{Tree(\texttt{t},\{1,2,3\} \cup \{3z|z \in \alpha\} \cup \{3\texttt{x}\}),Snap(\texttt{s},\{1,2,3\}),Iter(\texttt{it},\beta')\})\}$
```
8:   lc = it.hasNext();
```
$\{\alpha@\beta = [1,2,3] \wedge \texttt{lc} = (|\beta'| \neq 0) \wedge \beta = \texttt{x} :: \beta' \wedge$
$H(\{Tree(\texttt{t},\{1,2,3\} \cup \{3z|z \in \alpha\} \cup \{3\texttt{x}\}),Snap(\texttt{s},\{1,2,3\}),Iter(\texttt{it},\beta')\})\}$
```
9: }
```
$\{H(\{Tree(\texttt{t},\{1,2,3,6,9\})\},Snap(\texttt{s},\{1,2,3\})\})\}$

Figure 10.1: Snapshotable tree client code verification

structure with a non-empty list ($x :: \alpha$). The list head is returned and the model of the Iter structure is updated to the remainder of the list.

10.3.3 Client Code Verification

To verify the client code from Section 10.2.2 we assume we are given a class C such that ITree C H holds for some H and then verify the client code under the precondition $\{H(\{Tree(\texttt{t},\{\})\})\}$.

Figure 10.1 gives a step-by-step proof of the client code from Section 10.2.2, with client code lines to the left and their postconditions to the right.

After inserting some items (line 1) to the tree, the model contains these items, {1, 2, 3}. In line 2, a snapshot `s` of the tree `t` is created. The invariant H now consists of the Tree structure and a Snap structure containing the same elements. For the client the abstract structures

are disjoint, but in an implementation, they will be realized using sharing. Indeed, for the A1B1 implementation, the concrete heap will be as shown in Figure 10.2a, where all the nodes are shared between the tree and the snapshot.

In line 3 an iterator `it` over the snapshot `s` is created. To apply the call rule of the `iterator` method, only the Snap structure is taken into account, the rest (the Tree structure) is framed out inside of H (via appropriate instantiation of ϕ in the `iterator` specification). The result is that an Iter structure is constructed, whose model contains the same values as the model of the snapshot, but converted to an ordered list. We introduce the loop condition `lc` in line 4, and again use abstract framing to call `hasNext`.

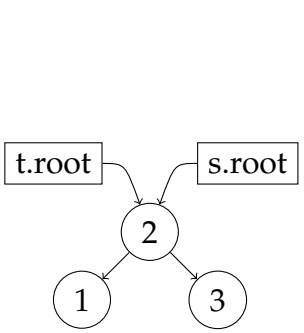
Lines 5–9 contain a while loop with loop condition `lc`. The loop invariant splits the iteration list $[1, 2, 3]$ into the list α containing the elements already iterated over and the list β containing the remainder. The loop variable `lc` is false iff β is the empty list. The invariant H contains the Tree structure whose model is the initial set {1, 2, 3} joined with the set of the elements of α, each multiplied by 3. H also contains the Iter and the Snap structures.

We omit detailed explanation of the remaining lines of verification.

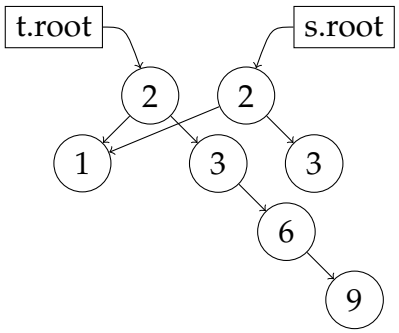
Note that in the final postcondition, the client sees two disjoint structures (axiom (e) is used to forget the empty iterator), but in the A1B1 implementation, the concrete heap will involve sharing, as shown in Figure 10.2b. Only the left subtree is shared by the tree and the snapshot; the root and right subtree were unshared by the first call to `add` in the loop.

In summary, we have shown the following theorem, which says that given any H and any classes C and IC satisfying the ITree and Iterator interface specifications, the client code satisfies its specification. The postcondition states that snapshot `s` contains 1, 2 and 3, and tree `t` contains additionally 6 and 9.

Theorem 1. $\forall H.\forall C.\forall IC.ITree\ C\ H \wedge Iterator\ IC\ H$
$\vdash \{H(\{Tree(\mathtt{t}, \{\})\})\}$
`client_code`
$\{H(\{Tree(\mathtt{t}, \{1,2,3,6,9\}), Snap(\mathtt{s}, \{1,2,3\}))\}$



(a) Heap after snapshot construction



(b) Live heap after execution of the client code

Figure 10.2: Heap layout of the trees during execution of client code.

10.4 Implementation A1B1

In this section we show the partial correctness verification of the A1B1 implementation with respect to the abstract specification from the previous section. This involves defining a concrete H and showing that the methods satisfy the required specifications for this concrete H. The development has been formally verified in Coq (as has the client program verification above).

The Coq formalization uses a shallow embedding of higher-order separation logic, developed for verification of OO programs using interfaces. See [12].

Invariant H is radically different depending on whether snapshots of the tree are present or not. The reason is that method `add` mutates the existing tree if there are no snapshots present, see Section 10.5 for details. Here we focus on the case where snapshots are present.

The A1B1Tree class stores its data in three fields: the `root` node, a boolean field `isSnapshot`, indicating whether it is a snapshot, and a field `hasSnapshot`, indicating whether it has snapshots. The stamp field mentioned in Section 10.2.3 is only required for iterators over the tree and so not further discussed here.

The Node class is a nested class of the A1B1Tree with three fields, `item` containing its value, and a handle to the right (`rght`) and left (`left`) subtree.

In the following we use standard separation logic connectives, in particular the separating conjunction $*$and the points to predicate $\mapsto$.

We now define our concrete H and also the realization of the abstract structures. We first explain the realization of Tree and Snap; the Iter structure is described in Section 10.4.1. Recall that ϕ ranges over finite sets of abstract structures (Tree, Snap, Iter), with exactly one Tree structure, and recall that H, given a ϕ, returns a separation logic predicate. The definition of H is:

$$H(\phi) \triangleq \exists\sigma.wf(\sigma) \wedge heap(\sigma) * \sigma \vDash \phi$$

Here σ is a finite map of type ptr $\rightarrow$ ptr $\times \mathbb{Z} \times$ ptr, with ptr being the type of Java pointers (handles), corresponding to the Node class. The map σ must be well-formed (pure predicate $wf(\sigma)$), which simply means that all pointers in the codomain of σ are either `null` or in the domain of σ.

The *heap* function maps σ to a separation logic predicate, which describes the realization of σ as a linked structure in the concrete heap, starting with $\top$:

$$\begin{array}{rl} heap(\sigma) \triangleq fold\ (\lambda \mathtt{p}\ n\ Q.\ match\ \ n\ with\ (\mathtt{pl}, v, \mathtt{pr}) \Rightarrow & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{p.left} \mapsto \mathtt{pl} * & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{p.item} \mapsto v * & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{p.rght} \mapsto \mathtt{pr} * & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Q) & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \top & \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \sigma & \end{array}$$

Finally, we present the definition of $\sigma \vDash \phi$ (we defer the definition of $\sigma \vDash \{Iter(\_,\_)\}$ to the following subsection):

$$\begin{array}{rl} \sigma \vDash \phi \uplus \psi \triangleq & \sigma \vDash \phi * \sigma \vDash \psi \\ \sigma \vDash \{Tree(\mathtt{ptr}, \tau)\} \triangleq & \exists \mathtt{p}.Node(\sigma, \mathtt{p}, \tau) \wedge \\ & \mathtt{ptr.root} \mapsto \mathtt{p} * \\ & \mathtt{ptr.isSnapshot} \mapsto \mathtt{false} * \\ & \mathtt{ptr.hasSnapshot} \mapsto \mathtt{true} \\ \sigma \vDash \{Snap(\mathtt{ptr}, \tau)\} \triangleq & \exists \mathtt{p}.Node(\sigma, \mathtt{p}, \tau) \wedge \\ & \mathtt{ptr.root} \mapsto \mathtt{p} * \\ & \mathtt{ptr.isSnapshot} \mapsto \mathtt{true} * \\ & \mathtt{ptr.hasSnapshot} \mapsto \mathtt{false} \end{array}$$

The spatial structure of all the nodes is covered by $heap(\sigma)$ so $\sigma \vDash \phi$ just needs to describe the additional heap taken up by Tree, Snap, and Iter structures.

The pure *Node* predicate is defined inductively on τ below. It is used to express that τ is the finite set of items reachable from p in σ.

$$\begin{aligned} Node(\sigma, \mathtt{p}, \tau) \triangleq\ & \big(\mathtt{p} = \mathtt{null} \wedge \tau = \{\}\big) \vee \\ & \big(\mathtt{p} \in dom(\sigma)\ \wedge \exists \mathtt{pl}, v, \mathtt{pr}.\ \sigma[\mathtt{p}] = (\mathtt{pl}, v, \mathtt{pr}) \wedge \\ & \exists \tau_l, \tau_r . \tau = \tau_l \cup \{v\} \cup \tau_r \wedge \\ & (\forall x \in \tau_l . x < v) \wedge (\forall x \in \tau_r . x > v) \wedge \\ & Node(\sigma, \mathtt{pl}, \tau_l) \wedge Node(\sigma, \mathtt{pr}, \tau_r)\big) \end{aligned}$$

The sortedness constraint (a strict total order) in the *Node* predicate enforces implicitly that σ has the right shape: σ cannot contain cycles and the left and right subtrees must be disjoint. The set τ is split into three sets, one with strictly smaller elements (τ_l), the singleton v and with strictly bigger elements (τ_r).

10.4.1 Iterator

The TreeIterator class implements the Iterator interface. It contains a single field, `context`, which is a stack of Node objects.

The constructor of the TreeIterator pushes all nodes on the leftmost path of the tree onto the stack. The method `next` pops the top node from the stack and returns the value held in that node. Before returning, it pushes the leftmost path of the node's right subtree (if any) onto the stack. The method `hasNext` returns true if and only if the stack is empty.

The verification of the iterator depends on the following specification of a stack class, generic in C. The specification is parametrized over a representation type T and existentially over a representation predicate SR (of type classname $\rightarrow$ (val $\rightarrow$ T $\rightarrow$ HeapAsn) $\rightarrow$ val $\rightarrow$ T^* $\rightarrow$ HeapAsn). The second argument is the predicate P (of type val $\rightarrow$ T $\rightarrow$ HeapAsn) , which holds for every stack element. This specification is kept in the style of [96], although we use a different logic.

class Stack<C> {

$\top$	`new()`	$SR\ C\ P$ `ret` nil
$SR\ C\ P$ `this` α	`empty()`	`ret` $= (\alpha =$ nil$) \wedge SR\ C\ P$ `this` α
$SR\ C\ P$ `this` $\alpha * P$ `x` $t \wedge$ `x` $: C$	`push(x)`	$SR\ C\ P$ `this` $(t :: \alpha)$
$SR\ C\ P$ `this` $(t :: \alpha)$	`pop()`	P `ret` $t * SR\ C\ P$ `this` α
$SR\ C\ P$ `this` $(t :: \alpha)$	`peek()`	P `ret` $t *$ $(\forall u.P$ `ret` $u \mathrel{-\!\!*} SR\ C\ P$ `this` $(u :: \alpha))$

(a) $P\ v\ t \vdash P'\ v\ t \implies SR\ C\ P\ v\ \alpha \vdash SR\ C\ P'\ v\ \alpha$

}

For the purpose of specifying the iterators over snapshotable trees, we instantiate the type T with $\mathbb{Z}^*$; the model of a node on the stack is a list of integers. Intuitively, this list corresponds to the node value and the element list of its right subtree. The iterator is modelled as a list that is equal to the concatenation of the elements of the stack. We also require that the topmost element of the stack is nonempty (if present). This intuition is formalized in the interpretation of the Iter structure, where SR is a representation predicate of a stack:

$$\sigma \vDash \{Iter(\mathtt{p}, \alpha)\} \triangleq \exists \mathtt{st}.\ \mathtt{p.context} \mapsto \mathtt{st} * \exists \beta. stack\_inv(\beta, \alpha) \wedge SR\ \mathtt{Node}\ (NS\ \sigma)\ \mathtt{st}\ \beta.$$

To make this definition complete, we provide the definitions of *stack_inv*, which connects the representation of the stack with the representation of the iterator, and the definition of the NS predicate.

$$stack\_inv(xss, ys) \triangleq ys = \mathsf{concat}(xss) \wedge \begin{cases} \top & \text{iff } xss = \mathsf{nil} \\ xs \neq \mathsf{nil} & \text{iff } xss = xs :: xss' \end{cases}$$

$$NS\ \sigma\ \mathtt{node}\ \alpha \triangleq Node(\sigma, \mathtt{node}, \tau) \wedge\ \alpha = [\{x \in \tau | x \geq \mathtt{node.item}\}]$$

These definitions, along with an assumption that SR is the representation predicate of Stack (i.e., fulfills all the method specifications and axioms of *Stack_spec*) suffice to show the correctness of Iter-dependent methods. The axiom present in *Stack_spec* is needed to preserve iterators if some new memory is added to σ: it allows us to replace $(NS\ \sigma)$ with $(NS\ \sigma')$ as a representation predicate of stack objects under certain side conditions.

10.5 On the Verification of Implemented Code

We now give an intuitive description of how the A1B1 implementation was verified, given the concrete H defined above. We verified the complete implementation in Coq but only discuss the `add` method here. We used Kopitiam [79] to transform the Java code into SimpleJava, the fragment represented in Coq.

Method `add` calls method `addRecursive` with the root node to insert the item into the binary tree, respecting the ordering. Method `addRecursive`, shown below, must handle several cases:

- if there are no snapshots present, then
  - if the item `x` is already in the tree, then the heap is not modified.
  - if the item `x` is not in the tree, then a new node is allocated and destructively inserted into the tree.
- if there are snapshots present, then
  - if the item `x` is already in the tree, then the heap is not modified.
  - if the item `x` is not in the tree, then a new node is allocated and every node on the path from the root to the added node is replicated, so that the snapshots are unimpaired.

The implementation of `addRecursive` walks down the tree until a node with the same value, or a leaf, is reached. It uses the call stack to remember the path in the tree. If a node was added, either the entire path from the root to the added node is duplicated (if snapshots are present) or the handles to the left or right subtree are updated (happens destructively exactly once, the parent of the added node updates its left or right handle, previously pointing to `null`):

```
Node addRecursive (Node node, int item, RefBool updated) {
  Node res = node;
3 if (node == null) {
    updated.value = true;
    res = new Node(item);
```

```
6     } else {
        if (item < node.item) {
          Node newLeft = addRecursive(node.left, item, updated);
9         if (updated.value && this.hasSnapshot)
            res = new Node(newLeft, node.item, node.rght);
          else
12          node.left = newLeft;
        } else if (node.item < item) {
          Node newRght = addRecursive(node.rght, item, updated);
15        if (updated.value && this.hasSnapshot)
            res = new Node(node.left, node.item, newRght);
          else
18          node.rght = newRght;
        } //else item == node.item so no update
      }
21    return res;
    }
```

We now show the pre- and postcondition of `addRecursive` for the two cases where snapshots are present. If the item is already present in the tree, the pre- and postcondition are equal:

$$\begin{gathered}\{\texttt{updated.value} \mapsto \texttt{false} * \texttt{this.hasSnapshot} \mapsto \texttt{true} * \\ \mathit{heap}(\sigma) * \mathit{wf}(\sigma) \wedge \mathit{Node}(\sigma, \texttt{node}, \tau) \wedge \texttt{item} \in \tau\} \\ \texttt{addRecursive(node, item, updated)} \\ \{\texttt{updated.value} \mapsto \texttt{false} * \texttt{this.hasSnapshot} \mapsto \texttt{true} * \\ \mathit{heap}(\sigma) * \texttt{ret} = \texttt{node}\}\end{gathered}$$

The postcondition in the case that the item is added to the tree extends the map σ to σ', for which the heap layout and the well-formedness condition must hold. The Node predicate uses σ' and the finite set is extended with `item`:

$$\begin{gathered}\{\texttt{updated.value} \mapsto \texttt{false} * \texttt{this.hasSnapshot} \mapsto \texttt{true} * \\ \mathit{heap}(\sigma) * \mathit{wf}(\sigma) \wedge \mathit{Node}(\sigma, \texttt{node}, \tau) \wedge \texttt{item} \notin \tau\} \\ \texttt{addRecursive(node, item, updated)} \\ \{\texttt{updated.value} \mapsto \texttt{true} * \texttt{this.hasSnapshot} \mapsto \texttt{true} * \\ \exists \sigma'. \sigma \subseteq \sigma' \wedge \mathit{heap}(\sigma') * \mathit{wf}(\sigma') \wedge \mathit{Node}(\sigma', \texttt{ret}, \{\texttt{item}\} \cup \tau)\}\end{gathered}$$

The call to `addRecursive` inside of `add` is verified for each specification of `addRecursive` independently.

To summarize Sections 10.4 and 10.5, we state the following theorem, which says there exists an H that given a stack fulfilling the stack specification, the TreeIterator class meets the Iterator specification and the A1B1 class meets the ITree specification, and the constructor for the A1B1Tree establishes the H predicate.

Theorem 2. $\exists H.\mathit{Stack\_spec} \vdash \mathit{Iterator\ TreeIterator}\ H \wedge \mathit{ITree}\ \texttt{A1B1}\ H \wedge \{\top\}\ \texttt{A1B1Tree}()\ \{H(\{\mathit{Tree}(\texttt{ret}, \{\})\})\}$
The client code, independently verified, can be safely linked with the A1B1 implementation!

10.6 Related Work

Malecha and Morrisett [78] presented a formalization of a Ynot implementation of B-trees with an iterator method. In their case, the iterator and the tree also share data in the concrete heap. However, they can only reason about "single-threaded" uses of trees and iterators: their specification of the iterator method transforms the abstract tree predicate into an abstract iterator predicate, which prohibits calling tree methods until the client turns the iterator back into a tree. In our setup, we have one tree, but multiple snapshots and iterators, and the tree can be updated after an iterator has been created. To permit sharing between a tree and an iterator, Malecha and Morrisett use fractional permissions, where we use the H predicate. They work in an axiomatic extension of Coq, whereas our proofs are done in a shallowly embedded program logic, since our programs are written in an existing programming language (Java).

Dinsdale-Young et al. [43] present another approach to reasoning about shared data structures, which gives the client a fiction of disjointness. Roughly speaking, they define a new abstract program logic for each module (they can be combined) for abstract client reasoning. Their approach allows one to give a client specification similar to ours, but without using the H and with the abstract structures (Tree / Snap / Iter) being predicates in the (abstract) program logic. This has the advantage that one can use ordinary framing for local reasoning.

Dinsdale-Young et al. [42] also presented an approach to reasoning about sharing. Sharing can happen in certain regions, and the module implementor has to define a protocol that describes how data in the shared region can evolve. What corresponds to our abstract structures can now be seen as separation logic predicates and thus one can use ordinary framing for local reasoning.

In both approaches [43] and [42] the module implementor has more proof obligations than in our approach: In [43] he must show that the abstract operations satisfy some conditions related to the realization of the abstract structures in the concrete heap. In [42] she must show related properties phrased in terms of certain stability conditions.

Compared to the work of Dinsdale-Young et al., our approach has the advantage that it is arguably simpler, with no need to introduce new separation (or context) algebras for the modules. That is why we could build our formalization on an implementation of standard separation logic in Coq.

Rustan Leino made a solution for a custom implementation of this data structure (A1B1) using Dafny [74]. Dafny verifies that if a snapshot is present, the nodes are shared and not mutated by the tree operations. His solution does not (yet) verify the content of the tree, snapshots or iterators. Our verification specifies the concrete heap layout. Dafny does not support abstract specification due to the lack of inheritance. The trusted code base is different: Dafny relies on Boogie, Z3 and the CLR, whereas our proof trusts Coq.

10.7 Conclusion and Future Work

We have presented snapshotable trees as a challenge for formalized modular reasoning about mutable data structures that use sharing extensively, and given an abstract specification of the ITree interface. Moreover, we have presented a formalization of the A1B1 implementation of snapshotable trees.

The overall size of the formalization effort is roughly 5000 lines[2] of Coq code and it takes 2 hours to `Qed` the proofs. This is quite big com-

[2]Update from July 2013: By using new tactics of the Charge! framework [11], the size of the formalization is now 2200 lines. The actual proof lines are reduced by a factor of 5.5, from 3325 lines to 670 lines.

pared to other formalization efforts of imperative programs in Coq, such as Hoare Type Theory / Ynot [87, 88]. The main reason is that we are working in a shallowly embedded program logic for a Java-like language, whereas Hoare Type Theory / Ynot is an axiomatic extension of Coq. Thus our formalization includes both the operational semantics of the Java subset and the soundness theorems for the program logic; also, Java program variables cannot simply be represented by Coq variables.

We also plan to verify the even subtler implementations A1B2, A2B1 and A2B2, which are expected to provide further insight into the challenges of dealing with shared mutable data and unobservable state changes. Through those more complex applications of separation logic we hope to learn more about desirable tool support, including how to automate the "obvious" reasoning that currently requires much thought and excessive amounts of proof code. Although we have not formally verified these implementations yet, we are fairly certain they would match the interface specification presented in Section 10.3. In all four implementations the tree is conceptually separate from its snapshots, which is the property required by the interface, and the invariant H allows us to describe the heap layout very precisely, using techniques shown in Section 10.4.

Finally, we would like to explore how to combine the advantages of our approach and those of Dinsdale-Young's approach discussed above.

10.8 Appendix

We define here the ITree and the Iterator interface specification as Coq definitions, as well as the Stack class. We use the name *SPred* for the finite set of abstract structures containing exactly one Tree structure and any number of Snap and Iter structures.

The notation $\widehat{f}$ lifts the function f such that it operates on expressions rather than values. A detailed explanation of the notation and of lifting can be found in [12].

$$
\begin{aligned}
\mathit{Stack\_spec} \triangleq\ & \forall T : \mathit{Type}. \\
& \exists SR : \mathit{classname} \to (\mathit{val} \to T \to \mathit{HeapAsn}) \to \mathit{val} \to T^* \to \\
& \quad \mathit{HeapAsn}. \\
& (\forall C : \mathit{classname}.\ \forall P : \mathit{val} \to T \to \mathit{HeapAsn}. \\
& \quad \texttt{Stack::new}() \mapsto \{\top\}\_\{\texttt{r}.\ \widehat{SR}\ C\ P\ \texttt{r}\ \mathsf{nil}\} \\
& \wedge (\forall \alpha : T^*.\ \texttt{Stack::empty}(\texttt{this}) \mapsto \\
& \quad \{\widehat{SR}\ C\ P\ \texttt{this}\ \alpha\}\_ \\
& \quad\ \{\texttt{r}.\ \widehat{SR}\ C\ P\ \texttt{this}\ \alpha \wedge \texttt{r} = (\alpha = \mathsf{nil})\}) \\
& \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::push}(\texttt{this}, \texttt{x}) \mapsto \\
& \quad \{\widehat{SR}\ C\ P\ \texttt{this}\ \alpha * \widehat{P}\ \texttt{x}\ t \wedge \texttt{x} : C\}\_ \\
& \quad\ \{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\}) \\
& \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::pop}(\texttt{this}, \texttt{x}) \mapsto \\
& \quad \{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\}\_\{\texttt{r}.\ \widehat{P}\ \texttt{r}\ t * \widehat{SR}\ C\ P\ \texttt{this}\ \alpha\}) \\
& \wedge (\forall \alpha : T^*.\ \forall t : T.\ \texttt{Stack::peek}(\texttt{this}, \texttt{x}) \mapsto \\
& \quad \{\widehat{SR}\ C\ P\ \texttt{this}\ (t :: \alpha)\}\_\{\texttt{r}.\ \widehat{P}\ \texttt{r}\ t * \\
& \quad\ (\forall u : T.\ \widehat{P}\ \texttt{r}\ u \mathrel{-\!\!*} \widehat{SR}\ C\ P\ \texttt{this}\ (u :: \alpha))\})) \\
& \wedge (\forall C : \mathit{classname}.\ \forall P, P' : \mathit{val} \to T \to \mathit{HeapAsn}. \\
& \quad (\forall v : \mathit{val}.\ \forall t : T.\ (P\ v\ t \vdash P'\ v\ t)) \implies \\
& \quad\ \forall v : \mathit{val}.\ \forall \alpha : T^*.\ (SR\ C\ P\ v\ \alpha \vdash SR\ C\ P'\ v\ \alpha))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ITree} \triangleq \lambda C : \mathit{classname}.\ \lambda H : \mathcal{P}_{fin}(\mathit{SPred}) \to \mathit{HeapAsn}.\\
&\quad (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{contains}(\texttt{this},\texttt{x}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this},\tau)\} \uplus \phi)\}\_\\
&\qquad\quad \{\texttt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this},\tau)\} \uplus \phi) \wedge \texttt{r} = (\texttt{x} \in \tau)\})\\
&\quad \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{contains}(\texttt{this},\texttt{x}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Snap}}(\texttt{this},\tau)\} \uplus \phi)\}\_\\
&\qquad\quad \{\texttt{r}.\ \widehat{H}(\{\widehat{\mathsf{Snap}}(\texttt{this},\tau)\} \uplus \phi) \wedge \texttt{r} = (\texttt{x} \in \tau)\})\\
&\quad \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{add}(\texttt{this},\texttt{x}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this},\tau)\} \uplus \phi)\}\_\\
&\qquad\quad \{\texttt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this},\{\texttt{x}\} \cup \tau)\} \uplus \phi) \wedge \texttt{r} = (\texttt{x} \notin \tau)\})\\
&\quad \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{snapshot}(\texttt{this}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this},\tau)\} \uplus \phi)\}\_\\
&\qquad\quad \{\texttt{r}.\ \widehat{H}(\{\widehat{\mathsf{Tree}}(\texttt{this},\tau), \widehat{\mathsf{Snap}}(\texttt{r},\tau)\} \uplus \phi)\})\\
&\quad \wedge (\forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{iterator}(\texttt{this}) \mapsto\\
&\qquad \{\widehat{H}(\{\widehat{\mathsf{Snap}}(\texttt{this},\tau)\} \uplus \phi)\}\_\{\texttt{r}.\ \exists IC : \mathit{classname}.\ \mathit{Iterator}\ IC\ H \wedge\\
&\qquad\quad \texttt{r} : IC \wedge \widehat{H}(\{\widehat{\mathsf{Snap}}(\texttt{this},\tau), \widehat{\mathsf{Iter}}(\texttt{r},[\tau])\} \uplus \phi)\})\\
&\quad \wedge (\forall v : \mathit{val}.\ \forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\qquad (H(\{\mathit{Tree}(v,\tau)\} \uplus \phi) \implies v : C) \wedge\\
&\qquad\quad (H(\{\mathit{Snap}(v,\tau)\} \uplus \phi) \implies v : C))\\
&\quad \wedge (\forall v : \mathit{val}.\ \forall \tau : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\qquad (H(\{\mathit{Snap}(v,\tau)\} \uplus \phi) \vdash H(\phi)))\\
&\quad \wedge (\forall v : \mathit{val}.\ \forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\qquad (H(\{\mathit{Iter}(v,\alpha)\} \uplus \phi) \vdash H(\phi)))\\
&\quad \wedge (\forall v : \mathit{val}.\ \forall \tau, \tau' : \mathcal{P}_{fin}(\mathbb{Z}).\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\\
&\qquad \tau = \tau' \implies (H(\{\mathit{Tree}(v,\tau)\} \uplus \phi) \vdash H(\{\mathit{Tree}(v,\tau')\} \uplus \phi)))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Iterator} \triangleq\ & \lambda C : \mathit{classname}.\ \lambda H : \mathcal{P}_{fin}(\mathit{SPred}) \to \mathit{HeapAsn}. \\
& (\forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{hasNext}(\texttt{this}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Iter}}(\texttt{this}, \alpha)\} \uplus \phi)\}\_ \\
& \quad\quad \{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Iter}}(\texttt{this}, \alpha)\} \uplus \phi) \wedge \mathtt{r} = (\alpha \neq \mathsf{nil})\}) \\
& \wedge (\forall x : \mathbb{Z}.\ \forall \alpha : \mathbb{Z}^*.\ \forall \phi : \mathcal{P}_{fin}(\mathit{SPred}).\ C{::}\texttt{next}(\texttt{this}) \mapsto \\
& \quad \{\widehat{H}(\{\widehat{\mathsf{Iter}}(\texttt{this}, x{::}\alpha)\} \uplus \phi)\}\_ \\
& \quad\quad \{\mathtt{r}.\ \widehat{H}(\{\widehat{\mathsf{Iter}}(\texttt{this}, \alpha)\} \uplus \phi) \wedge \mathtt{r} = x\})
\end{aligned}
$$

Chapter 11

Functional Verification of a Point Location Algorithm

Abstract

In this paper we verify the functional correctness of a solution to the point location problem. The point location problem is of geometric nature: in a plane, which is divided into disjoint polygonal regions, find the smallest region which contains a queried point. Several solutions have space complexity linear in the number of edges that form the regions and provide a logarithmic query time also in the number of edges. We implement and outline a functional correctness proof of one solution which meets these time and space complexity by using shared mutable state. We use a higher-order separation logic for verification of its correctness.

11.1 Introduction

The point location problem [41, Chapter 6] is well known in geometry. Consider a plane, which is subdivided into regions. The task is to find the region which contains the query point. The subdivision of the plane by n edges into regions is known ahead of time, and the query time may not exceed $O(\log n)$. Additionally a space constraint of $O(n)$ is imposed.

In this paper we contribute a full functional specification and verification of a real-world solution to the point location problem, which

meets the time and space complexity requirements. We use separation logic to verify the correctness of our implementation. We describe our Java implementation in detail, which is based on an implementation in C# that serves as a showcase of the collection library C5 [66]. We point out where we refactored the code to simplify its verification. A side effect of these modifications was that the code is easier to understand, thus they have also been applied to the original implementation. The full Java source is available at `http://www.itu.dk/people/hame/PlanarPoint.java`.

This paper first describes the point location problem in Section 11.2, followed by the solution we are verifying in Section 11.3. Afterwards we describe the implementation in detail in Section 11.4. We give a specification in Section 11.5, which we use to verify the query implementation in Section 11.6. In Section 11.7 we describe the related work. Finally, we conclude by describing the modifications to the original code in Section 11.8 and describe future work in Section 11.9.

11.2 Point Location Problem

The point location problem occurs frequently in real world applications: given a political or geographical map and a query point by its coordinates, find the smallest region (or country) of the map containing the queried point. A map is a planar subdivision of the plane, where the regions are disjoint.

Another appearance of this problem is a web browser - if a user moves the mouse pointer to any point and presses the mouse button, the underlying HTML element should be activated: a link should be followed, text highlighted, etc.

Both occurrences might have to handle a lot of data — a huge map or a complex website — and to be interactive for user input, the query time should be minimal.

As running example we use a very simple division of the plane, shown in Figure 11.1. The plane is divided into the region `A` with a nested region `B`. The algorithm should return region `B` for the queried point `q`.

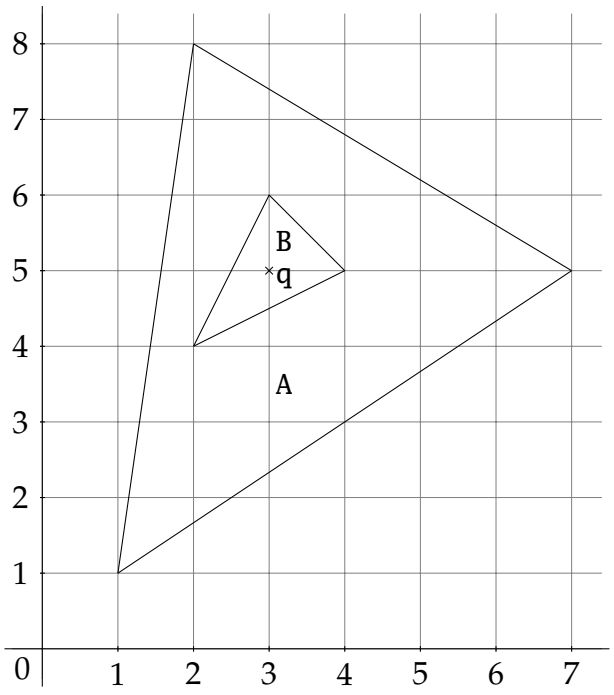


Figure 11.1: Nested regions: the outer region A contains completely the inner region B. Anything outside of the region A does not belong to a region. The query point q belongs to the inner region B.

11.2.1 Problem Statement

We decompose the point location problem into two phases. The first phase, named *build*, is static and receives the planar subdivision as n line segments. The second phase is *query*, which is dynamic and receives queries. A query consists of a point in the plane specified by its coordinates. The smallest region which contains the queried point must be returned.

We narrow down the problem slightly by considering only a two dimensional plane. We constrain the list of line segments: no line segment may intersect with another one, no line segment is vertical (starts and ends at the same x-coordinate), and the set of line segments must form closed regions. The containing region for a query point which is exactly on a line segment is unspecified. The limitation that no line segment is vertical can be overcome by rotation with a small angle, because we have a finite set of line segments.

The point location problem is extensively studied in the literature, the book *Computational Geometry: Algorithms and Applications* [41] de-

votes a complete chapter [41, Chapter 6] to the problem. It includes a comprehensive overview of solutions and possible extensions of the problem, such as making the planar subdivision dynamic as well, or scaling up to more than two dimensions.

A naive implementation would build polygons out of the n line segments during the *build* phase and, during the *query* phase, check whether the queried point is inside each polygon. This requires $O(n)$ space and also $O(n)$ query time during the query phase. We will ignore the time consumption during the build phase, since this happens only once. Different enhancements trade reduced query time for increased space usage. One such enhancement is to divide the plane by drawing a vertical line through the starting and ending point of each line segment. This partitions the plane into vertical *slabs*. For each slab a binary search tree containing all line segments which intersect this slab is constructed. We store the x-coordinate of the slab together with a binary tree of its line segments. To query for a concrete point, first a lookup of the slab tree to the left is done, then the line segment above and below the query point are looked up in the binary tree. While the query time reduces to $O(\log n)$, the space usage increases to $O(n^2)$. In this paper, we look at a further improvement of this approach that reduces the space usage.

The point location problem has a long history in computational geometry. Several solutions in the literature show that for n line segments a linear space usage of $O(n)$ and logarithmic query time of $O(\log n)$ can be accomplished for the two dimensional case. Solutions with this complexity are described in [41, Chapter 6]: the chain method [47] based on segment trees; the triangulation refinement method [63]; the randomized incremental method [86]; and the snapshotable binary tree method [101]. In this paper we will describe the solution based on snapshotable binary trees [46], which we specified and verified in previous research [82].

11.3 Solution

We illustrate the algorithm of the solution [101] in more detail by continuing with our example from Figure 11.1.

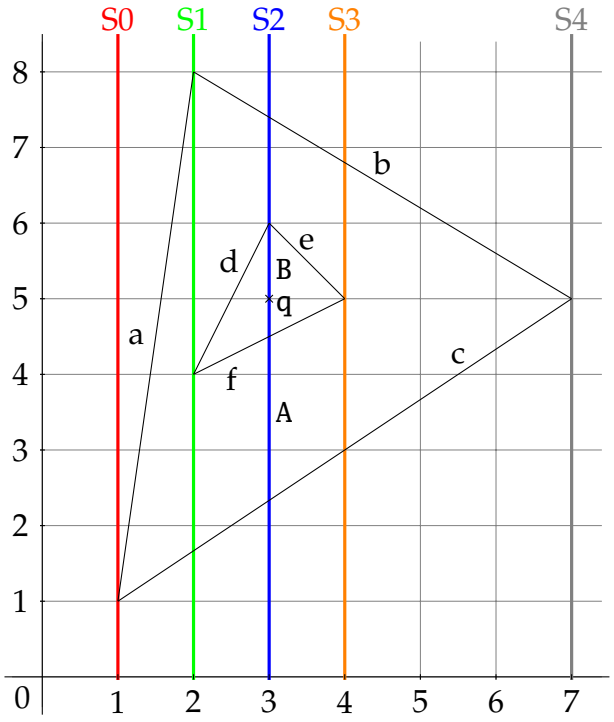
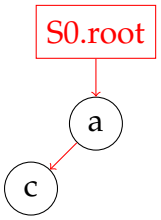


Figure 11.2: Partition of Figure 11.1 into five slabs.

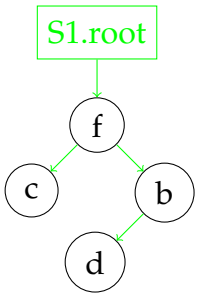
11.3.1 Build Phase

During the *build* phase, the planar subdivision is partitioned into vertical slabs, shown in Figure 11.2. A slab is created for the starting and ending point of each line segment.

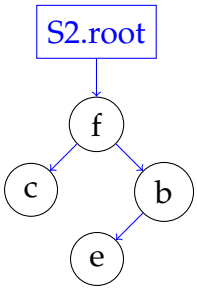
For each slab we build a *line segment tree*, which is a binary tree containing the line segments which intersect with the slab. Line segments which end at the slab are not included, but line segments which start at the slab are included. The non-empty line segment trees (all but S4) are shown in Figure 11.3, where nodes are the named line segments from Figure 11.2. Each line segment contains the information which region is above and below it. These line segment trees are stored in a *slab tree*, also a binary tree. The slab tree uses the x-position of the line segment trees as key for lookup and insertion.



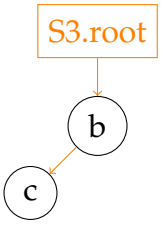
(a) Tree of slab S0 of Figure 11.2.



(b) Tree of slab S1 of Figure 11.2.



(c) Tree of slab S2 of Figure 11.2.



(d) Tree of slab S3 of Figure 11.2.

Figure 11.3: Slab trees of Figure 11.2.

Optimization: Meeting Space Requirement

When we construct the line segment trees for each slab separately, the space consumption is $O(n^2)$. To meet the required $O(n)$ space consumption, we instead share subtrees between the line segment trees by using snapshotable trees [46]. These have an amortized space consumption of $O(n)$. The actual heap of our shared trees is shown in Figure 11.4, where colours indicate separate trees. Snapshotable trees use *copy-on-write* and a persistent readonly handle can be taken at any time. Each node stores a third (backup) child reference together with a count of the amount of snapshots, thus every node may belong to several snapshots and the tree itself. For each slab, a snapshot of the line segment tree is taken and stored inside the slab tree. In our example the node b has to be copied to b′ for the tree S3, since the backup reference of node b is already used for node e. Note that in the figure each tree is indicated by a different color, but the actual heap layout uses less references: the line segment

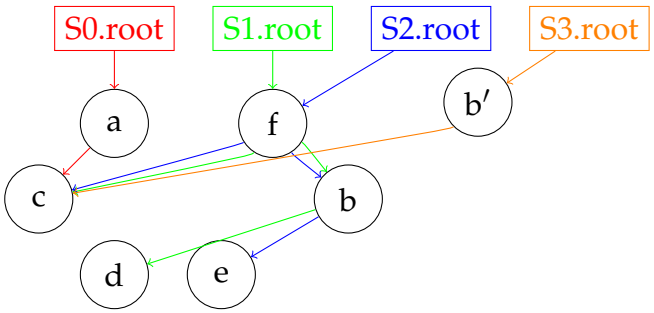


Figure 11.4: Heap layout of the snapshotable trees of Figure 11.3.

snapshot for slab S1 and slab S2 share references from node f to node c and from node f to node b.

11.3.2 Query Phase

In the *query* phase, for a given query point first the nearest line segment tree to the left of the queried point is located. This is done by a lookup in the slab tree, which consumes $O(\log m)$ time for m slabs.

Consider our query point `q` from Figure 11.1. This query point lies exactly on the tree S2, which is then used. The tree S2 is shown in Figure 11.3c.

Now, the two closest line segments to the query point `q` are looked up. This is done by a tree search, where the comparison checks whether the line segment is above or below the query point, and preserves the closest line segments in a record.

Our query point `q` is clearly above the line segment f, thus the right subtree is used, and the closest above is set to f. The queried point is also below b and below e. The closest line segment below `q` is f, the closest line segment above `q` is e. The algorithm compares the region above f and below e, which must be identical. In our example the region is in both cases `B`, which is then returned.

The query time is a lookup in the binary slab tree, which consumes $O(\log m)$ time for m slabs, and another binary tree search for the closest line segments, which takes $O(\log n)$ time, for n line segments. Due to the condition that line segments form closed polygons, $m \leq n$ holds. The total time consumed for query is therefore $O(\log m + \log n)$, which is the stipulated $O(\log n)$.

11.4 Implementation

An implementation of the point location algorithm using snapshotable trees was developed by Kokholm and Sestoft as part of the C5 library [66] for C#. We used this implementation as a basis and manually translated it to Java, and refactored it with verification in mind.

We now briefly recapitulate the snapshotable binary tree interface, which is similar to the one shown in our earlier paper [82]. We describe the code of both the `Build` and the `Query` method. In Section 11.6 we verify the `Query` method. Additionally, we describe the implementation challenge of line segments and their ordering.

11.4.1 Snapshotable Binary Tree

Conceptually, a snapshotable binary tree is a normal binary search tree with the additional operation `snapshot`. This operation returns a persistent readonly copy of the tree in constant time. The implementation exploits possible sharing between the tree and its snapshots, by copying only when a node is inserted or deleted (*copy-on-write*).

The Java standard library confluences monotonicity with partial ordering. From a logic perspective we want to distinguish between elements which have a partial ordering and elements which behave monotonically regarding a set of other elements. We introduce the interface `Monotonic<A>` with the single method `compareTo` to achieve this separation. The interface `Comparable<A>` extends our `Monotonic<A>` without introducing additional methods. Thus, the implementor does not see a difference between these interfaces, but the logical view distinguishes these interfaces.

The interface `ITree` is defined as follows, where `A` must have a partial ordering by extending the `Comparable<A>` interface, whereas the `cut` operation requires an object which behaves monotonically on the tree:

```
public interface ITree<A extends Comparable<A>> {
  public boolean add (A item);
  public boolean remove (A item);
  public CutResult<A> cut (Monotonic<A> cmp);
  public ITree<A> snapshot ();
}
```

The operations have the following effect:

- `tree.add(item)` inserts the object `item` in the tree and returns `true` if the object was not already there. Otherwise, it does nothing and returns `false`.
- `tree.remove(item)` removes the object `item` from the tree and returns `true` if the object was there. Otherwise, it does nothing and returns `false`.
- `tree.cut(cmp)` receives a callback `cmp`, which has the single method `cmp.compareTo(A a)`. It is monotonic in `A`. If the tree contains an object `x`, for which `cmp.compareTo(x)` returns 0, this `x` is returned in both fields of the `CutResult<A>` structure shown below. Otherwise, if there is an object in the tree for which `cmp.compareTo` returns 1, the greatest such object is returned in the `high` field. Otherwise, if there is an object in the tree for which `cmp.compareTo` returns -1, the smallest such object is returned in the `low` field. Both `low` and `high` can be `null`: if the tree is empty, or if the comparator returns 1 (or -1) for all tree elements.
- `tree.snapshot()` returns a readonly snapshot of the given tree in constant time. Updates to the given tree will not affect the snapshot. Only the operation `cut` is supported for a snapshot.

The `CutResult` class is shown next. It contains a `low` and a `high` field, representing the results of the cut.

```
class CutResult<A> {
  A low;
  A high;
}
```

The interface `ITree<A>` provides only the operations we need for the actual implementation of the point location algorithm in the interest of simplicity. The previously verified implementation [82] is not polymorphic over tree elements, and does not include neither a `remove` nor a `cut` operation. Instead it implements and verifies an `iterator` operation by extending Java's `Iterable` interface.

11.4.2 PointLocation class

We will describe the concrete implementation of the `PointLocation` class. In this class the slab tree of type `ITree<Pair<ITree<LineSegment>>>` is stored in the `htree` field. Our `Pair<A>` has two fields: a primitive integer and a polymorphic `A`. We use the primitive integer for the x-position, and a `ITree<LineSegment>`, which contains a line segment tree, such as S0 to S3 from Figure 11.4. A `LineSegment` is a four-tuple record of start position, end position, region above it, and region below it.

The class `PointLocator` provides two methods: `Build` receives a list of `LineSegment` and builds up the slab tree; the method `Query` receives a query point and returns the region identifier which contains it. This identifier is an integer value in our case, where -1 means no region.

We consider line segments and query points to start and end at integral values, we will explain this in more detail in Section 11.4.3.

```
class PointLocation {
  ITree<Pair<ITree<LineSegment>>> htree;

  void Build (List<LineSegment> l) { ... }

  int Query (int x, int y) { ... }
}
```

Build Method

The `Build` method is presented in Figure 11.5. It receives the list `l` of line segments and constructs a `SortedList` out of them. This sorted list `sl` contains triples: the x-position, a list of line segments ending at this position, and a list of line segments starting at this position.

The field `htree` contains the slab tree. It is initialized to an empty tree. The line segment tree `vtree` is also initialized to an empty tree. At every slab, a snapshot of the line segment tree is stored into the slab tree. An empty snapshot of the line segment tree is inserted at negative infinity (lines 8–9) into the slab tree. This is used for queries left of the leftmost x-position of any slab.

In the main loop, for each triple of the sorted list (lines 12 and 33), at the concrete position x the ending line segments are removed from

```
void Build (List<LineSegment> l) {
  assert(htree == null);
  SortedList<LineSegment> sl = l.toSortedList();
  htree = new Tree<ITree<LineSegment>>();
  ITree<LineSegment> vtree = new Tree<LineSegment>();

  //put a snapshot at "negative infinity"
  htree.add(new Pair<ITree<LineSegment>
    (Integer.MIN_VALUE, vtree.snapshot()));

  SortedNode cur = sl.head;
  while (cur != null) {
    //remove ending line segments
    Node<LineSegment> c = cur.endLineSegments.head;
    while (c != null) {
      boolean rem = vtree.remove(c.val);
      assert(rem == true);
      c = c.next;
    }

    //insert starting line segments
    Node<LineSegment> d = cur.startLineSegments.head;
    while (d != null) {
      boolean add = vtree.add(d.val);
      assert(add == true);
      d = d.next;
    }

    //take snapshot and insert (x, snapshot) into htree
    ITree<LineSegment> snap = vtree.snapshot();
    htree.add(new Pair<ITree<LineSegment>>(cur.x, snap));

    cur = cur.next;
  }
}
```

Figure 11.5: Implementation of method `Build`.

```
int Query (int x, int y) {
  assert(htree != null);

  //Find the closest tree to the left of our query point
  TreeComp tc = new TreeComp(x);
  CutResult<Pair<ITree<LineSegment>>> cr = htree.cut(tc);
  ITree<LineSegment> p = cr.low.getValue();

  //Find the closest line segments above and below our query point
  PointLineComparer<LineSegment> plc =
    new PointLineComparer<LineSegment>(x, y);
  CutResult<LineSegment> res = p.cut(plc);

  int r = -1;
  if (res.high != null && res.low != null) {
    //we found a cut, the regions must be identical:
    //  below the upper line segment and
    //  above the lower line segment
    assert(res.high.below() == res.low.above());
    r = res.low.above();
  }
  return r;
}
```

Figure 11.6: Implementation of method Query.

the line segment tree (lines 13–19), the starting line segments are added to the line segment tree (lines 21–27), and a pair of x-position and its corresponding snapshot is inserted into the slab tree (lines 29–31).

Query Code

The procedure Query looks up the region for a given point, provided by its x- and y-position. The implementation of the method Query is shown in Figure 11.6.

First, the closest line segment tree to the left of the query point is looked up (lines 4–7) in the slab tree. We use the cut operation for this, along with the tc object, which is an instance of the TreeComp class. A TreeComp compares its stored x-position with the first projection of the given pair.

Afterwards, the line segments directly below and above the queried point are located by using a `PointLineComparer` and the `cut` operation on the line segment tree (lines 9–12). A `PointLineComparer` class stores an x- and y-position, and judges whether a given line segment is above or below the point by using the determinant, described in more detail in the next subsection.

If there is a line segment above the queried point and one below the queried point (line 15), we ensure that they agree on the region between them (line 19) and return that region (line 20).

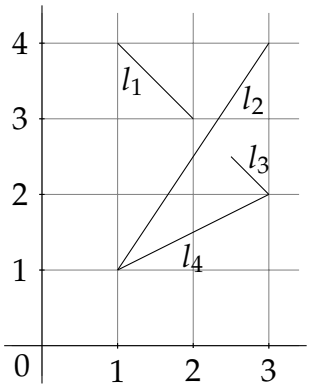
11.4.3 Line Segments

We discuss two aspects of line segments: their representation and how we establish an ordering relation.

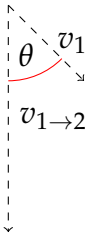
We limit the implementation to use integral values for the start and end position of line segments. Additionally, queries can only be done at integral positions. The reason is that floating point numbers, while used widely to represent real numbers, do not have a concise mathematical model. Although there are libraries for Coq to reason about floating point numbers, we are more interested in the point location algorithm than in number representations. Additionally, when comparing floating point numbers, representation and rounding errors have to be compensated for by using a small epsilon. This would complicate the verification and make it verbose.

Each line segment is a quadruple of data: the start point, the end point, the regions to the right and the left of the line segment. Left and right are defined as if we were at the start point, looking into the direction of the end point. The x-position of the start point is always lower than or equal to the x-position of the end point. We disallow vertical line segments. Since a line segment is a directed vector, the definition which region is right and left of it is unique. One of the regions may be `null`, if the line segment is at the outer border of a polygon. For example, the line segment a in Figure 11.2 has no region to the left, and the region `A` to the right.

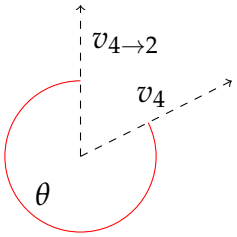
We define an order relation on line segments. The intuition behind this order relation is that a line segment is "greater" if it is above another line segment. The order relation is used for insertion and removal



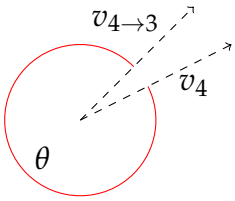
(a) Four line segments l_1, l_2, l_3, and l_4 in a plane. The line segment l_1 is above all other line segments. The line segment l_2 is above l_3, which is above l_4. The line segments l_1 and l_3 are on the same line, the comparison between these can be done solely based on their x coordinates.



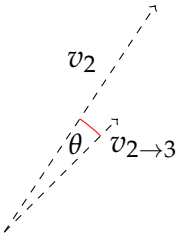
(b) Line segment l_1 is above line segment l_2, because the angle θ between the vector v_1, which corresponds to the line segment l_1, and the vector $v_{1\to2}$, which corresponds to the vector from the start point of l_1 to the start point of l_2, is smaller than 180°.



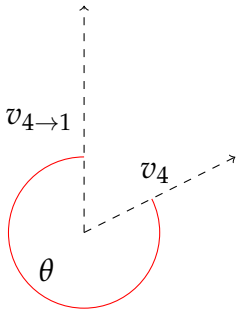
(c) Line segment l_4 is below line segment l_2. The line segments l_4 and l_2 share a common start point, thus end points are used for comparison. The angle θ between the vector v_4 and the vector $v_{4\to2}$ from the end point of l_4 to the end point of l_2 is greater than 180°.



(d) Line segment l_4 is below line segment l_3, because the angle θ between the vector v_4, which corresponds to l_4, and the vector $v_{4\to3}$ from the start point of l_4 to the start point of l_3 is greater than 180°.



(e) Line segment l_2 is above line segment l_3, because the angle θ between the vector v_2, which corresponds to l_2, and the vector $v_{2\to3}$ from the start point of l_2 to the start point of l_3 is smaller than 180°.



(f) Line segment l_4 is below line segment l_1, because the angle θ between the vector v_4, which corresponds to l_4, and the vector $v_{4\to1}$ from the start point of l_4 to the start point of l_1 is greater than 180°.

Figure 11.7: Four line segments, presented in Figure 11.7a, and their ordering.

of line segments into the slab tree, and also for locating the nearest line segments during the *query* phase. Recall that line segments may not intersect. If two line segments were allowed to intersect, our order relation would depend on the x-position - to the left of the intersection one line segment is above the other, whereas to the right of the intersection the other is above the one. We define an ordering relation which does not depend on the x-position of non-intersecting line segments by using the cross product. Four line segments l_1 to l_4 and their ordering are shown in Figure 11.7. For our ordering relation we compute the cross product between the vector, which is the line segment with the smaller x-position and the vector from the start point of this line segment and the start point of the *other* line segment. If the line segments have a common start point, the end points are used instead. In Figure 11.7b this is illustrated between the line segments l_1 and l_2 from Figure 11.7a. The vector v_1 of l_1 and the vector $v_{1\to 2}$ from the start point of l_1 to the start point of l_2 are shown, and the cross product between them is computed. The cross product between two vectors a and b is defined as: $a \times b \equiv |a| * |b| * sin\ \theta * n$, where θ is the angle between a and b, $|a|$ is the magnitude of the vector a, and n is the unit vector perpendicular to the plane containing a and b. The sign of the cross product depends solely on $sin\ \theta$: if θ is less than 180°, the cross product is positive, otherwise negative. If the cross product is positive, the chosen line segment is above the other one. Otherwise the chosen line segment is below the other one.

We also use the cross product to find out whether a given point is above or below a given line segment in the `TreeComp` class. The cross product is analogous to the determinant in the two dimensional space. We compute the cross product of the line segment vector and the vector from the query point to the start of the line segment, and get the angle between them as a result. If that angle is above 0, the point is above the line segment. Otherwise, it is below.

11.5 Specification

In this section we give a specification for the `PointLocator` class and the `ITree` interface. This is used as a prerequisite for Section 11.6, in which

we outline the correctness proof of the implementation of `Query`. We define representation predicates which describe the heap layout by using separation logic. Separation logic allows us to describe disjoint parts of the heap modularly. We remarked in our earlier verification [82] that the predicate for a tree without snapshots is completely different from a tree with snapshots. We need to have knowledge about the line segment tree and all its snapshots to describe the heap layout of the nodes, because the nodes can be shared between the line segment tree and its snapshots. In contrast to the line segment tree, our slab tree is a binary search tree without snapshots, thus we use a common representation predicate for binary trees to describe its heap shape. Although the slab tree and the line segment tree use the same implementation, their representation predicates are different. In this section we specify the `PointLocator` class, and use different predicates for both trees.

The high-level informal specification of the *build* procedure is that it receives a set of non-intersecting line segments which form non-overlapping closed polygons. It associates these polygons with the *PL* (PointLocation) predicate. During the *query* phase the predicate *PL* is used to lookup the polygon containing the queried point. When using geometric reasoning and intuition, verification thereof is straightforward. This is different when using a proof assistant which does not have geometric intuition.

In this section we formalise the informal specification, using our previous work, in which we verified the snapshotable tree data structure [82] with higher-order separation logic. Our specification is slightly adapted to cope with changing demands. We also use a generic tree interface which abstracts over the tree elements by adapting Svendsen et al. [110].

11.5.1 `PointLocator` specification

We first specify the `PointLocator` class, which contains two methods, `Build` and `Query` and a constructor. It contains the field `htree`, which stores a reference to the slab tree, initialized to `null`.

The *PL* predicate is defined on a Java reference `ptr`, a set of polygons p and asserts a predicate on the heap, where $\mapsto$ is the separation logic

points-to predicate and $*$ the separating conjunction.

$$\begin{aligned} PL\ \texttt{ptr}\ p \triangleq\ & (\texttt{ptr.htree} \mapsto \texttt{null}\ \wedge\ p = \emptyset)\ \vee \\ & \Big(\exists\ \texttt{tl}.\ \ \texttt{ptr.htree} \mapsto \texttt{tl} * \exists\ ls,\ ts.\ HTree\ \texttt{tl}\ (zip\ (posls)\ ts) * \\ & \quad ls = \bigcup ts * wf\ p\ ls\Big) \end{aligned}$$

There are two cases: either the set of polygons p is empty, in which case the reference in the `htree` field points to `null`, or p is not empty, and then the reference in the `htree` field points to some location `tl`, for which the *HTree* predicate is defined, which we describe later. The model of the slab tree is a list of pairs, zipped together from the list of x positions (gathered by the function *pos*) and the sets of line segments ts. The union ls of all line segment sets ts must be well-formed (wf) with regards to the set of polygons p.

The function *pos* takes a set of line segments and returns their x value of starting and ending positions:

$$\begin{aligned} pos\ \ \emptyset\ &\triangleq \emptyset \\ pos\ \ \{ls\} \cup rt\ &\triangleq \{x(begin\ ls), x(end\ ls)\} \cup pos\ rt \end{aligned}$$

The set of line segments ls is well-formed, defined by the predicate wf, if no two segments intersect, there are no vertical line segments, all line segments are connected, and together they form the set of closed polygons p:

$$\begin{aligned} wf\ p\ ls \triangleq\ & non\text{-}intersecting\ ls\ \wedge non\text{-}vertical\ ls\ \wedge \\ & connected\ ls\ \wedge closed\ ls\ p \end{aligned}$$

With these definitions we specify the `PointLocator` class, using a notation where we define the pre- and postconditions before and after the method names. All bindings which are not explicitly quantified are implicitly universally quantified, the special variable `r` in the postcondition is bound to the return value of the method.

```
class PointLocator {
 ⊤              new PointLocator()   PL r ∅

 PL this ∅ ∗
 List l ls ∗    Build(l)             PL this ps
 wf ps ls

 PL this ps     Query(x, y)          ((x,y) ∈ ps[r] ∨ r = −1) ∗ PL this ps ∗
                                     (∀p : ps. (p = ps[r] ∨ (x,y) ∉ p ∨ ps[r] ⊃ p))
}
```

- The constructor returns a `PointLocator` object which satisfies the *PL* predicate with the empty set (∅) of polygons.
- The `Build` method receives a list of line segments `l`, whose model *ls* is well-formed regarding some polygons *ps*. It also requires a *PL* predicate that relates the instance to the empty set of polygons. It ensures the *PL* predicate is true with regards to the set of polygons *ps* formed by the given line segments *ls*.
- The `Query` method receives a point (x, y), and returns an index to the smallest polygon which contains the point or -1 if no such polygon exists. The order relation on polygons is defined by inclusion. For each point (x, y), the finite set of polygons containing (x, y) has a smallest element. This follows from the fact that no two polygons overlap unless one is completely contained in the other.

11.5.2 Predicate *HTree*

The predicate *HTree* describes the heap shape of the slab tree that contains the forest of line segment trees. We use an existentially quantified data structure global invariant H (which we introduced in our earlier work [82]) to describe the heap shape of the forest of line segment trees.

$$HTree\ \texttt{ptr}\ m \triangleq \exists\, \phi, H.\ HTree'\ \texttt{ptr}\ m\ \phi * H(\phi)$$

The predicate $HTree'$ describes the heap layout of the binary slab tree: each node consists of a reference to its content, a pair of the x-position and a reference to its line segment tree snapshot (v), its left child (l), and its right child (r).

The model m, a list of integer-tree pairs, is destructed into a left part m_l and a right part m_r, and a pair of an x-position and a tree model ts in the middle. We use @ as the append operator. The x positions in the left part are strictly smaller and the ones in the right part are strictly larger. This is formalised by using the first projection (Π_1) of m_l and m_r.

The set ϕ of abstract structures contains the set of abstract structures of the left (ϕ_l) and right (ϕ_r) subtree, and a *Snap* structure with the reference v and the model ts.

The left and right subtrees are either empty, when `l`, respectively `r` point to `null`, or described recursively by the $HTree'$ predicate.

$$
\begin{aligned}
HTree'\ \texttt{ptr}\ m\ \phi \triangleq\ & \exists\ m_l,\ m_r,\ x,\ ts.\ \ m = m_l @ [(x, ts)] @ m_r \wedge \\
& (\forall y : m_l.\ \Pi_1 y < x) \wedge (\forall y : m_2.\ \Pi_1 y > x)\ * \\
& \exists\ \mathtt{v}.\ \ \texttt{ptr.node} \mapsto (x,\ \mathtt{v})\ * \\
& \exists\ \phi_l, \phi_r.\ \ \phi = \{Snap(\mathtt{v}, ts)\} \uplus \phi_l \uplus \phi_r\ * \\
& \exists\ \mathtt{l}.\ \ \texttt{ptr.left} \mapsto \mathtt{l}\ * \\
& ((\mathtt{l} = \texttt{null} \wedge \phi_l = \emptyset) \vee HTree'\ \mathtt{l}\ m_l\ \phi_l)\ * \\
& \exists\ \mathtt{r}.\ \ \texttt{ptr.right} \mapsto \mathtt{r}\ * \\
& ((\mathtt{r} = \texttt{null} \wedge \phi_r = \emptyset) \vee HTree'\ \mathtt{r}\ m_r\ \phi_r)
\end{aligned}
$$

11.5.3 Snapshotable Tree Specification

We briefly describe the representation predicates for the snapshotable tree. The representation predicate R_{val} (not shown) is defined on a Java reference and a model T, which describes the heap layout of the Java reference. The elements of T must have an ordering relation c, which meets the common ordering laws (anti-symmetry, reflexivity and transitivity).

While separation logic reasons about disjoint heaps, our tree implementation shares nodes between the tree and its snapshots. This sharing is not observable by a client. In order to achieve that, we introduce the global invariant H under which we define abstract structures *Tree*, *Snap* and *CR*. A client can consider each of these abstract structure to be disjoint from the rest of the abstract structures, and frame out the parts she does not need. Since the H is existentially quantified, the client has no knowledge of how an implementation defines H. The implementation has a global view on the tree with all its snapshots and can define which

parts of the abstract structure are shared in the concrete heap. The abstract structures consist of a reference to the tree and a model, which is an ordered finite set, containing the elements of the tree. The `ITree` interface is specified as follows:

`interface` `ITree {`

$H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi) *$ $R_{val}\ \texttt{x}\ e$	`add(x)`	$\texttt{r} = e \notin \tau *$ $H(\{\mathit{Tree}(\texttt{this}, \{e\} \cup \tau)\} \uplus \phi)$
$H(\{\mathit{Tree}(\texttt{this}, \{e\} \cup \tau)\} \uplus \phi) \wedge$ $R_{val}\ \texttt{x}\ e \in H$	`remove(x)`	$\texttt{r} = \texttt{true} \wedge$ $H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi) *$ $R_{val}\ \texttt{x}\ e$
$H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi) \wedge$ $e \notin \tau * R_{val}\ \texttt{x}\ e$	`remove(x)`	$\texttt{r} = \texttt{false} \wedge$ $H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi) *$ $R_{val}\ \texttt{x}\ e$
$H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi)$	`snapshot()`	$H(\{\mathit{Tree}(\texttt{this}, \tau),\ \mathit{Snap}(\texttt{r}, \tau)\} \uplus \phi)$
$H(\{\mathit{Snap}(\texttt{this}, \tau)\} \uplus \phi) *$ $R_{cmp}\ \texttt{cmp}\ c * \mathit{monotonic}\ c\ \tau$	`cut(cmp)`	$H(\{\mathit{CR}(\texttt{r}, c, \tau), \mathit{Snap}(\texttt{this}, \tau)\} \uplus \phi) *$ $R_{cmp}\ \texttt{cmp}\ c * \mathit{monotonic}\ c\ \tau$
$H(\{\mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi) *$ $R_{cmp}\ \texttt{cmp}\ c * \mathit{monotonic}\ c\ \tau$	`cut(cmp)`	$H(\{\mathit{CR}(\texttt{r}, c, \tau), \mathit{Tree}(\texttt{this}, \tau)\} \uplus \phi) *$ $R_{cmp}\ \texttt{cmp}\ c * \mathit{monotonic}\ c\ \tau$

`}`

The specification of the operations on the tree are as follows:

- `add` requires a *Tree* abstract structure and inserts the model of the given `x` into the finite set τ.
- `remove` is split into two cases: either the given `x` is in the tree, or not. Both require a *Tree* abstract structure and a model e of the given `x`. In the first case, the model τ of the tree contains the model e. After this operation, τ no longer contains e, and the return value is `true`. In the other case, the model τ does not contain e. The return value is `false` and the tree was not modified.
- `snapshot` requires a *Tree* abstract structure and returns a *Snap* abstract structure with the same model τ.
- `cut` requires a *Snap* or *Tree* abstract structure and a comparator which is monotonic regarding the element set τ. What is returned

is the abstract structure *CR* describing the two tree elements which are closest to where the comparator flips from -1 to 1.

The representation predicate R_{cmp} is not shown here. It contains the state of the comparator, and relates the comparator to the modelled ordering c. We now define the predicate *monotonic*, used for the `cut` method specification. Intuitively the ordering of the comparator model is the same as the ordering of the modelled objects.

$$\mathit{monotonic}\ c\ \tau \triangleq \forall\ a,\ b : \tau.\ a \leq b \rightarrow c\ a \leq c\ b$$

Similarly to our previous verification [82], we define a concrete H. Recall that ϕ ranges over finite sets of abstract structures (*Tree*, *Snap*, *CR*); and that H returns a separation logic predicate, given a ϕ:

$$H(\phi) \triangleq \exists \sigma.\mathit{wf}'(\sigma) \wedge \mathit{heap}(\sigma) * \sigma \vDash \phi$$

The σ is a finite map from reference to a triple (reference $\times$ R_{val} $\times$ reference), which represents the Node structure. The map σ must be well formed, which means that all references in the codomain of σ are either `null` or in the domain of σ. The function *heap* maps σ to a separation logic predicate, which describes the realization of σ as a linked structure in the concrete heap.

Finally, we present the definition of $\sigma \vDash \{CR(\_,\_,\_)\}$, which models the `CutResult` class. It asserts that the two objects are indeed the closest above and below of the cut. All objects must be either the selected one above the cut, above it, or below the cut. Analogously for the selected below the cut.

$$\begin{aligned}\sigma \vDash CR(\texttt{ptr}, c, \tau) \triangleq\ & \big((\exists\ \texttt{hi}.\ \texttt{ptr.high} \mapsto \texttt{hi} * \exists\ h.\ R_{val}\ \texttt{hi}\ h \wedge c\ h > 0 \wedge \\ & \quad (\forall\ t : \tau.\ t = h \vee t > h \vee c\ t < 0)) \vee \\ & \quad (\texttt{ptr.high} \mapsto \texttt{null} \wedge \forall\ t : \tau.\ c\ t < 0)\big) \wedge \\ & \big((\exists\ \texttt{lo}.\ \texttt{ptr.low} \mapsto \texttt{lo} * \exists\ l.\ R_{val}\ \texttt{lo}\ l \wedge c\ l < 0 \wedge \\ & \quad (\forall\ t : \tau.\ t = l \vee t < l \vee c\ t > 0)) \vee \\ & \quad (\texttt{ptr.low} \mapsto \texttt{null} \wedge \forall\ t : \tau.\ c\ t > 0)\big)\end{aligned}$$

The specification of the comparators `TreeComp` and `PointLineComparer` is not shown here for conciseness. Each constructor of these returns a representation predicate R_{cmp} `this` c, which is monotonic regarding the tree objects (*monotonic* c τ).

11.6 Verification

In this section we use the specification developed in Section 11.5 to outline a proof of the correctness of the `Query` method, which we presented in Section 11.4.2.

We did not present a specification for the slab tree applying the `cut` method. This operation returns aliases to nodes of the tree, thus we have to apply a similar mechanism as to the snapshots, namely a data structure global invariant. Since the aliases do not escape the Query method, we neglect the exact details here.

We intermingle the source code with the symbolic heap and text describing verification steps. To distinguish, we prefix the source code always with a line number.
We start with the method precondition.

{ *PL* `this` p }

1: `int Query (int x, int y) {`
We first unpack (expand) the predicate *PL*.

{ (`this.htree` $\mapsto$ `null` $\wedge$ $p = \emptyset$)$\vee$
($\exists$`tl`. `this.htree` $\mapsto$ `tl` $*$ $\exists$ *ls*, *ts*. *HTree* `tl` (*zip* (*pos ls*) *ts*) $*$
$ls = \bigcup ts * wf\ p\ ls$) }

2: `assert(htree != null);`
We remove the first part of disjunction that would lead to an assertion failure.

{ $\exists$`tl`. `this.htree` $\mapsto$ `tl` $*$ $\exists$ *ls*, *ts*. *HTree* `tl` (*zip* (*pos ls*) *ts*) $*$
$ls = \bigcup ts * wf\ p\ ls$ }

3: `TreeComp tc = new TreeComp(x);`
A new object is instantiated, which is referred to by the local variable `tc`. The constructor returns some representation predicate which is monotonic regarding the slab tree elements (pairs of an integer and a line segment snapshot).

$\{\ R_{cmp}\ \texttt{tc}\ c * \mathit{monotonic}\ c\ (x, \tau) *$
$\exists \texttt{tl}.\ \texttt{this.htree} \mapsto \texttt{tl} * \exists\ ls,\ ts.\ \mathit{HTree}\ \texttt{tl}\ (\mathit{zip}\ (\mathit{pos}\ ls)\ ts) *$
$ls = \bigcup ts * \mathit{wf}\ p\ ls\ \}$

We unpack the predicate HTree to prepare for the cut.

$\{\ R_{cmp}\ \texttt{tc}\ c * \mathit{monotonic}\ c\ (x, \tau) *$
$\exists \texttt{tl}.\ \texttt{this.htree} \mapsto \texttt{tl} *$
$\exists\ ls,\ ts,\ \phi,\ H.\ \mathit{HTree}'\ \texttt{tl}\ (\mathit{zip}\ (\mathit{pos}\ ls)\ ts)\ \phi * H(\phi) *$
$ls = \bigcup ts * \mathit{wf}\ p\ ls\ \}$

4: `CutResult<Pair<Tree<LineSegment>>> cr = htree.cut(tc);`

The variable `cr` contains the result of the `cut` method, which is a `CutResult` of a pair of an integer and a line segment snapshot. We use the rule of consequence to remove predicates concerning the `tc` from our symbolic heap, which we do not need for the remaining proof.

$\{\ \mathit{CR}(\texttt{cr}, c, \tau) *$
$\exists \texttt{tl}.\ \texttt{this.htree} \mapsto \texttt{tl} *$
$\exists\ ls,\ ts,\ \phi,\ H.\ \mathit{HTree}'\ \texttt{tl}\ (\mathit{zip}\ (\mathit{pos}\ ls)\ ts)\ \phi * H(\phi) *$
$ls = \bigcup ts * \mathit{wf}\ p\ ls\ \}$

Next we unpack the *CR* predicate to access `cr.low`. For readability we do not list all formulae of the *CR* predicate, only those we need in later steps. It is important to note that the `Build` method puts a line segment snapshot at negative infinity, thus `cr.low` will never be a reference to `null`. The finite set of abstract structures ϕ contains at least a *Snap* at x'.

$\{\ \exists \texttt{tl}.\ \texttt{this.htree} \mapsto \texttt{tl} *$
$\exists\ ls,\ ts,\ \phi,\ H.\ \mathit{HTree}'\ \texttt{tl}\ (\mathit{zip}\ (\mathit{pos}\ ls)\ ts)\ \phi * H(\phi) *$
$\exists \texttt{lo}, x'.\ \texttt{cr.low} \mapsto (x', \texttt{lo}) * \mathit{Snap}(\texttt{lo}, \tau) \in \phi * x' \leq \texttt{x} *$
$ls = \bigcup ts * \mathit{wf}\ p\ ls\ \}$

We shorten the predicate slightly by rearranging and framing out parts we no longer need.

$\{\ \exists\ \texttt{lo},\ x'.\ H(\{\mathit{Snap}(\texttt{lo}, \tau)\}) * \texttt{cr.low} \mapsto (x', \texttt{lo}) * x' \leq \texttt{x}\ \}$

5: `ITree<LineSegment> p = cr.low.getValue()`

We replace the existentially quantified `lo` with the local variable `p`.

$\{\ H(\{\mathit{Snap}(\texttt{p}, \tau)\})\ \}$

6:
```
PointLineComparer<LineSegment> plc =
  new PointLineComparer<LineSegment>(x, y);
```

A new object is instantiated. The local variable `plc` refers to it, and the postcondition of the constructor contains a representation predicate and that it is monotonic regarding line segments.

$\{\ R_{cmp}\ \texttt{plc}\ c * \mathit{monotonic}\ c\ \tau * H(\{\mathit{Snap}(\texttt{p}, \tau)\})\ \}$

7: `CutResult<LineSegment> res = p.cut(plc);`

We call the method `cut` on the line segment tree which `p` refers to, and the point line comparer `plc`. This method returns a `CutResult`. We use the rule of consequence to shorten the symbolic heap for readability.

$\{\ H(\{\mathit{Snap}(\texttt{p}, \tau), \mathit{CR}(\texttt{res}, c, \tau)\})\ \}$

8: `int r = -1;`

A new local variable appears on the stack.

$\{\ r = -1 * H(\{\mathit{Snap}(\texttt{p}, \tau), \mathit{CR}(\texttt{res}, c, \tau)\})\ \}$

We unfold H to access the fields of the `CutResult` afterwards.

$\{\ r = -1 * \exists \sigma.\mathit{wf}'(\sigma) \land \mathit{heap}(\sigma) * \sigma \vDash \mathit{CR}(\texttt{res}, c, \tau) * \sigma \vDash \mathit{Snap}(\texttt{p}, \tau)\ \}$

We further unpack CR to access its internal fields.

$\{\ r = -1 * \exists \sigma.\mathit{wf}'(\sigma) \land \mathit{heap}(\sigma) * \sigma \vDash \mathit{Snap}(\texttt{p}, \tau) *$
$\big((\exists\ \texttt{hi}.\ \texttt{res.high} \mapsto \texttt{hi} * \exists\ h.\ R_{val}\ \texttt{hi}\ h \land$
$(\forall\ t : \tau.\ t = h \lor t > h \lor c\ t < 0)) \lor$
$(\texttt{res.high} \mapsto \texttt{null} \land \forall\ t : \tau.\ c\ t < 0)\big) \land$
$\big((\exists\ \texttt{lo}.\ \texttt{res.low} \mapsto \texttt{lo} * \exists\ l.\ R_{val}\ \texttt{lo}\ l \land$
$(\forall\ t : \tau.\ t = l \lor t < l \lor c\ t > 0)) \lor$
$(\texttt{res.low} \mapsto \texttt{null} \land \forall\ t : \tau.\ c\ t > 0)\big)\ \}$

9: `if (res.high != null && res.low != null) {`

We remove those assertions which cannot be valid due to this guard.

$\{\ r = -1 * \exists \sigma.\mathit{wf}'(\sigma) \land \mathit{heap}(\sigma) * \sigma \vDash \mathit{Snap}(\texttt{p}, \tau) *$
$(\exists\ \texttt{hi}.\ \texttt{res.high} \mapsto \texttt{hi} * \exists\ h.\ R_{val}\ \texttt{hi}\ h \land$
$(\forall\ t : \tau.\ t = h \lor t > h \lor c\ t < 0)) \land$
$(\exists\ \texttt{lo}.\ \texttt{res.low} \mapsto \texttt{lo} * \exists\ l.\ R_{val}\ \texttt{lo}\ l \land$
$(\forall\ t : \tau.\ t = l \lor t < l \lor c\ t > 0))\ \}$

10: `assert(res.high.below() == res.low.above());`

The intuition is that the two line segments in the `CutResult` object are neighbours. This is due to the specification of the `cut` method. Furthermore, since two line segments may not intersect, the region between those two line segments must be identical. The two line segments are

also the nearest ones to the queried point – thus the region between them must be the smallest one containing the queried point. A more formal proof has to take the well-formedness predicate into account: the line segments form non-overlapping closed polygons. As mentioned earlier, we assume that the queried point is not on a line segment; if we were to allow this, we would need to handle this in a special case here.

11: `r = res.low.above();`
The returned value `r` contains the region identifier above the lower line segment, which is the same as the one below the line segment above the point. The postcondition of the method holds: we pack the *PL* predicate, and the returned region is the smallest one containing the queried point.

$\{ (x,y) \in \mathtt{r} * PL\ \mathtt{this}\ p * (\forall\, q : p.\ (q = p[\mathtt{r}]\ \vee (x,y)\ \notin\ q\ \vee p[\mathtt{r}] \supset q)) \}$

12: `} else {`
The alternative branch, in which one or both line segments in the `CutResult` are `null`. This intuitively means that the queried point is either below or above all line segments of the tree, which might also happen if the line segment tree is empty. This means that there is no region containing the queried point. The postcondition is valid: the returned value `r` is -1, and none of the polygons *ps* contains the queried point (corresponding to the second disjunct). We can also successfully pack the *PL* predicate.

$\{ \mathtt{r} = -1 * PL\ \mathtt{this}\ p * (\forall\, q : p.\ (x,y)\ \notin\ q) \}$

13: `}`
Both branches of the conditional fulfill the postcondition of this method.

14: `return r; }`
We abstract the returned value `r` to fulfill the method postcondition from Section 11.5.

$\{ \mathtt{r}.\ \big((x,y) \in \mathtt{r} \vee \mathtt{r} = -1\big) * PL\ \mathtt{this}\ p *$
$(\forall\, q : p.\ (q = p[\mathtt{r}]\ \vee (x,y)\ \notin\ q\ \vee p[\mathtt{r}] \supset q)) \}$

We have outlined the verification of the Query method, using the our previous specification of the shapshotable tree data structure. The `Query` method returns the correct result: when the point is not contained in any region, the default value -1 is returned, otherwise a region containing

the queried point is returned, and furthermore this returned region is the smallest one containing the queried point.

11.7 Related Work

The verification of the point location problem builds on top of our previous research in verifying snapshotable trees [82]. We modified our earlier verification slightly to handle our demands. Instead of using integers as tree elements, we use a generic A here. We removed the implementation of the `Iterator` interface from the `ITree` interface, because this is not needed for the planar point location algorithm. The method `contains` was also removed from the `ITree` interface, but we added both the `remove` and the `cut` method. We believe that the main part of the proof, as well as the concrete instantiation of H can be reused for this modified interface and an implementation thereof.

The snapshotable tree data structure has also been verified by other researchers using different tools (Why3, Plural, Dafny).

Related to this data structure are other imperative data structures, like B-trees [78]. B-trees and their iterators share data in the concrete heap. But they only allow a single-threaded access to the data structure — the specification of the iterator method transforms the abstract tree predicate into an iterator predicate. This does not allow multiple snapshots and the tree predicate to be present at the same time, which we need in this verification. Their work uses fractional permissions to allow sharing between the tree and its iterator, where we use the H predicate.

11.8 Conclusion

We have demonstrated a specification and verification of the point location problem. Our implemented and verified solution is applicable to a real qne non-trivial algorithm. Our verification uses both parametric polymorphism (generics) and callbacks in particular, the method `cut` receives a callback to the comparator.

Since we started from real code, written without verification in mind, we have learned a lesson when preparing imperative programs for full functional verification: *smash the state*! Although we can reason about

shared mutable state with separation logic, we first refactored the program to use less mutable state, resulting in a clearer and easier to specify program.

We give evidence for our *smashing the state, one field at a time* slogan by showing two concrete examples where we refactored the original code: line segment comparison and the `Build` method. We removed the state of the line segment comparator by using the cross product for the ordering relation, as described in Section 11.4.3, and developed a `Build` method which does not need mutable state, described in Section 11.4.2. Hence we obtained more declarative and more comprehensible code as a side effect.

Line Segment Comparison The comparison of line segments, for insertion and removal into the snapshotable tree, formerly computed the y position at the concrete x position. This involves computing the slope of the line segment. The y position is a real number, represented as a floating point number in a computer. This representation is imprecise, and rounding errors occur. To compensate for both, a small epsilon was used during comparison. Additionally, the comparison contained some state, namely the current x position, which was modified when the line segment tree for the next position was built. Verifying such an ordering of line segments would involve reasoning about the monotonicity of the comparators. An ordering at position 2 should be valid at position 3. For general line segments this is not the case, but due to the fact that line segments do not intersect, it holds for our program. Additionally, the comparison used a field `compareToRight`, which tracked whether ending or starting line segments are compared. This was used as a tie breaker: if the y position of both line segments are equal, the slope of the line segments was compared - and depending whether the comparison happened at the start or endpoint, the higher or lower slope was considered to be greater or smaller.

Build Method Related to the comparison, the second piece of evidence is the `Build` method, which in the original code used an ordered list of pairs containing the endpoint and the line segment. The advantage was that only one loop was needed for this method, but an ordering on the endpoints was required. The ordering used the x position, then the y

position and as a tie braker the information whether the line segment was starting or ending at this position. In the `Build` phase a variable kept track of the x position in the last iteration. If the x position was changed, a snapshot was inserted into the slab tree.

Both code changes, the comparison method and the simplified `Build` method, were ported back to the C# implementation and merged into the main development branch of C5[1].

11.9 Future Work

Future work is in several areas: formalizing the model of the point location problem (geometry), unifying the `ITree` interface specification, and formalising the proof in Coq using Charge!.

We have two ideas to model geometry in Coq. The first is to look into Geoproof [89], which formalises Euclidean geometry inside of Coq. The second is to look deeper into the formalisation of the four colour theorem [50] proof. The latter is a well-publicized proof conducted using Coq. The theorem statement is: For any plane partitioned into regions, four different colours suffice to colour the regions so that no two adjacent regions share the same colour. We might be able to reuse parts of this formalisation, based on hypermaps, which define the adjacency and non-intersection properties of polygons.

The `ITree` interface is used both for the slab tree and the line segment tree, but the specification is different. At the moment, the slab tree is a standard binary search tree without snapshots, whereas several snapshots are taken from the line segment tree. We plan to generalize the specification for both use cases, which needs some more thought because of our global H invariant. The abstract structures *Tree* and *Snap* are defined under the invariant H, such that a user can use them disjointly, but the implementor of the `ITree` interface has a global view on the tree and all of its snapshots. The tree and its snapshots share parts of the heap, and we use the separating conjunction ($*$) between all nodes. This entails that abstract structures are not first-class citizens, which makes their nesting cumbersome to use.

To bypass a data-structure global invariant like H, two lines of research were established recently: Fictional separation logic [59] embeds

[1] `https://github.com/sestoft/C5/commits?author=hannesm`

the fiction of disjointness into the logic. Substructural types [69] combines dependent types with linear types, which are used to reason about ressources. Both areas of research might fit well for the problem presented in this paper.

Another area of future work is to mechanize our verification using our earlier development [82] within our higher-order separation logic framework Charge!. In order to formalise the presented verification within Charge!, we first need to extend Charge! to be able to reason about generics and delegates. Verification of generics and delegates using separation logic has been pioneered by Svendsen et al [110].

Our earlier verification development [82] does not include the methods `remove` and `cut`, which are required by the planar point implementation. Additionally, the implemented Java code uses path copy persistence, whereas for space and time complexity it should use node copy persistence and rebalance the tree if needed after insertion and removal of nodes.

Chapter 12

Verification of Snapshotable Trees using Access Permissions and Typestate

Originally published in: TOOLS 2012 [80]

Joint work with: Jonathan Aldrich; Carnegie Mellon University, Pittsburgh

Abstract

We use access permissions and typestate to specify and verify a Java library that implements snapshotable search trees, as well as some client code. We formalize our approach in the Plural tool, a sound modular typestate checking tool. We describe the challenges to verifying snapshotable trees in Plural, give an abstract interface specification against which we verify the client code, provide a concrete specification for an implementation and describe proof patterns we found. We also relate this verification approach to other techniques used to verify this data structure.

12.1 Introduction

In this paper we use access permission and typestate to formally verify snapshotable search trees in Plural [17, Chapter 6]. Snapshotable trees have been proposed as a verification challenge [82], because they contain abstract separation and internal sharing: the implementation uses sharing, while the user sees each tree and snapshot separately. The complete verified code is available at `http://www.itu.dk/people/hame/SnapTree.java`.

We only verify API compliance rather than full functional correctness in this paper. The protocol of the data structure is verified, rather than the tree content. The protocol is intricate, with internal sharing that is hidden from the client. The tree content could be modeled as a set, but in Plural no reasoning about sets is implemented.

We will first recapitulate the snapshotable tree verification challenge [82], typestate, and access permissions. Then we will briefly describe Plural and introduce our solution to the challenge.

To our knowledge this is the first formal verification of a tree data structure using access permissions and typestate. The verification of the Composite pattern [19], which consists of a tree data structure, used non-formalized extensions of Plural and was not formalized in Plural.

Snapshotable Search Trees A snapshotable search tree is an ordered binary tree with the additional method `snapshot`, which returns a handle to a readonly persistent view of the tree. Both the tree and the snapshot implement the same interface ITree. While the client can think of a tree and a snapshot as disjoint, the actual implementation requires that `snapshot` be computed in constant time. This is achieved by sharing the nodes between the tree and its snapshots. If a new node is inserted into the tree, the nodes are lazily duplicated (copy on write).

There are two implementation strategies, *path copy persistence* and *node copy persistence* [46]. While the former duplicates the entire path from the root node to the freshly inserted node, the latter has an additional handle in each node, which is used for the first mutation of the node.

```
public interface ITree extends Iterable<Integer> {
  public boolean contains(int x);
  public boolean add(int x);
  public ITree snapshot();
  public Iterator<Integer> iterator();
}
```

The methods of the ITree interface have the following effects:

- `contains` return true if the given item is in the tree, otherwise false.

- `add` inserts the given item into the tree. If the item was already present, this method does not have any effect and its return value is false, otherwise true.
- `snapshot` returns a readonly view of the current tree. Taking a snapshot of a snapshot is not supported.
- `iterator` returns an iterator of the tree's (or snapshot's) items.

We consider only iterators over snapshots for the remainder of the paper. There is no limit to the number of iterators over a snapshot. Iterators over a snapshot are valid even if the original tree is mutated.

Our client code uses this behaviour, and iterates over the snapshot while mutating the original tree:

```
void client (ITree t) {
  t.add(2); t.add(1); t.add(3);
  ITree s = t.snapshot();
  Iterator<Integer> it = s.iterator();
  while (it.hasNext()) {
    int x = it.next();
    t.add(x * 3);
  }
}
```

The client code adds some elements to a ITree, creates a snapshot, and iterates over the snapshot while it adds more items to the underlying tree. The client code is computationally equivalent to the original challenge [82], we do not introduce an unnecessary boolean variable for the loop condition.

Typestate Typestate systems [108] were developed to enhance reliability of software. A developer specifies the API usage protocol (a finite state machine) directly in the code. These protocols are statically checked. Empirical results [10] have shown that API protocol definitions occur three times more often than definitions of generics in object-oriented (Java) code. In Plaid, an upcoming programming language, typestate is a first-class citizen [109] and has been incorporated into the type system.

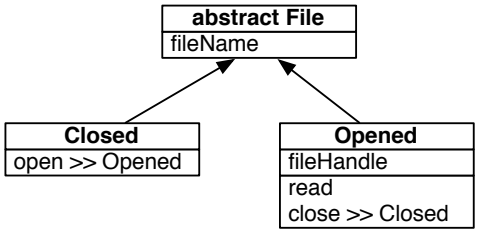


Figure 12.1: A typestate example showing a File class.

A motivating example for typestate is the *File* class, shown in Figure 12.1. Reading a file is only valid if it is open, thus the abstract *File* class has two states, *Opened* and *Closed*, and the method `read` is only defined in the *Opened* state. The method `open` (only defined in the *Closed* state) transitions the object from the *Closed* to the *Opened* state (indicated by $>>$), and vice versa for `close`.

This prevents common usage violations, like trying to read a closed file or opening a file multiple times.

Access Permissions A developer can annotate references with alias information [6] by using access permissions [17]. Access permissions are used for controlling the flow of linear objects. In the presented system there are five different permissions: exclusive access (*unique*), exclusive write access with others possibly having read access (*full*), shared write access (*share*), readonly access with others possibly having write access (*pure*) and immutable access in which no others can write either (*immutable*).

Boyland et al. [26] presented fractions to reason about permissions. This allow us to split and join permissions: for example a *unique* permission can be split into a *full* and a *pure*, which can later be merged back together.

Plural The Plural[1] tool does sound and modular typestate checking; it employs fractional permissions to provide flexible alias control.

[1] `https://code.google.com/p/pluralism/`

Plural was implemented as a plugin for Eclipse, on top of the Crystal framework[2]. It consists of a static dataflow analysis which tracks constraints about permissions in a lattice and infers local permissions.

A developer can annotate each interface with abstract states, specified by name. Interface methods can be annotated with pre- and post-conditions (required and ensured permissions and states).

Each class can be annotated with concrete states, which consist of a name and an invariant: a linear logic formula consisting of the access permission to a field in a specific state, or the (boolean or non-null) value of a field.

In a formula the standard linear logic conjunctions are available: *implies* ($\multimap$, written =>), *and* ($\otimes$, written $*$), *or* ($\oplus$, written +) and external choice (&).

Each state can be a refinement of another state; a state can be refined multiple times. We use this to refine the default *alive* state.

In order to access the fields of an object, the object must be unpacked, allowing temporary violations of the object's state invariants. Special care has to be taken to not unpack the same object multiple times (by using different aliases and permissions thereof), because this leads to unsoundness. Plural enforces the restriction that only a single object can be unpacked at a time. Before a method is called, all objects must be packed. Plural makes an exception to this rule for objects with unique permission, which is obviously sound since there cannot be any aliases to these objects.

Overview In Section 12.2 the interface specification and client code verification will be shown. Section 12.3 describes some proof patterns used in the verification of the actual implementation. In Section 12.4 we will describe related work and in Section 12.5 we conclude and present future work.

Our study is based on Plural, and indeed we observed certain low-level tool-specific artifacts (discussed in the conclusion) and the Plural-specific "ghost method" proof pattern. The main focus of this paper, however, including all other specification and proof patterns in Sections 12.2–12.3, is a high-level application of typestate and permission concepts to verify the tree and its clients. This may provide insights

[2] `https://code.google.com/p/crystalsaf/`

useful in other settings based on permissions [6, 26] and/or typestate [108, 109].

12.2 Interface Specification and Client Code Verification

The verification challenge is to give an abstract specification that does not expose implementation details, is usable by a client, and for each state an invariant can be specified by an implementor to verify her implementation.

We will first describe the specification of the interface ITree and Iterator, and afterwards we will show the verification of the client code using those specifications.

12.2.1 Interface ITree

We specify the interface ITree by having two disjoint typestates, *Tree* and *Snapshot*, which keep track whether the object is a tree or a snapshot of a tree. The `marker=true` annotation ensures that the state cannot change during the lifetime of an object. Both states refine the default state *alive*.

```
@States(refined="alive", value={"Tree", "Snapshot"}, marker=true)
interface ITree extends Iterable<Integer> {
  @Pure
  public boolean contains(int item);
  @Full(requires="Tree", ensures="Tree")
  public boolean add(int item);
  @Full(requires="Tree", ensures="Tree")
  @ResultPure(ensures="Snapshot")
  public ITree snapshot();
  @Pure(requires="Snapshot", ensures="Snapshot")
  @ResultUnique
  @Capture(param="underlying")
  public TreeIterator iterator();
}
```

The annotations are intuitive: the method `contains` requires a *pure* permission in any typestate, and returns the very same permission. The method `add` requires a *full* permission in the *Tree* state; the method

`snapshot` requires a *full* permission in the *Tree* state and the return value has a *pure* permission in the *Snapshot* state. The `iterator` method requires a *pure* permission in the *Snapshot* state, whereas the resulting iterator will have a *unique* permission. The `Capture` annotation indicates that `this` is captured by the returned `TreeIterator` object.

The access permissions and typestates formalize the informal constraints presented in the description of the ITree interface in Section 12.1.

12.2.2 Interface Iterator

Iterators have been specified previously in Plural [16], we include the specification for self-containedness of this paper. We follow similar ideas (namely a non-empty and empty state), whereas our implementation is different (see Section 12.3.5).

There are three states defined for an iterator, *NonEmpty*, *Empty* and *Impossible*, all refine *alive*. The last one is only for specifying the `remove` method which throws an exception in our implementation.

The method `next` requires *unique* permission to a *NonEmpty* iterator. The `hasNext` method requires *immutable* permission and if it returns true, the object is in the *NonEmpty* state, if false is returned, it is in the *Empty* state.

The need for a *unique* permission is due to recursive calls and Plural's restriction of having only a single unpacked object, mentioned in Section 12.1. We will discuss this in more detail when we show the iterator implementation in Section 12.3.5. This is a marginal drawback, since in practice iterators are used on the stack rather than shared via the heap.

This specification actually enforces that *hasNext* is called before each call to *next*, because otherwise the iterator is not known to be in the *NonEmpty* state.

```
@States(refined="alive", value={"NonEmpty", "Empty", "Impossible"})
interface TreeIterator extends Iterator<Integer> {
  @Unique(requires="NonEmpty")
  public Integer next();
  @Imm
  @TrueIndicates("NonEmpty")
  @FalseIndicates("Empty")
```

```
  public boolean hasNext();
  @Unique(requires="Impossible")
  public void remove();
}
```

12.2.3 Client Code Verification

The client code needs only a single annotation, that it has *full* permission in the *Tree* state of the given argument.

```
class ClientCode {
  @Perm(requires="full(#0) in Tree")
3 void client (ITree t) {
    t.add(2); t.add(1); t.add(3);
    ITree s = t.snapshot();
6   TreeIterator it = s.iterator();
    while (it.hasNext()) {
      int x = it.next();
9     t.add(x * 3);
    }
  }
12}
```

The method `client` adds the elements 1, 2 and 3 to the tree (line 4), creates a snapshot `s` (line 5) and an iterator `it` over the snapshot (line 6). The body of the while loop (lines 8 and 9) adds more elements to the original tree (line 9).

In this section we have demonstrated that the client code preserves the required permissions and states, using the given specification for the ITree and Iterator interfaces.

12.3 Proof Patterns and Verification of the Implementation

We have verified the A1B1 implementation [82], which does not implement rebalancing and uses path copy persistence: when a snapshot is present, the complete path from the root down to the newly inserted node is copied in a call to `add`. This ensures that add does not mutate any node that is shared between the snapshot and the tree.

The specifications of field getters, field setters, and constructors are omitted in the paper: they are straightforward, a field getter requires an *immutable* permission, a field setter a *full* permission and the constructor ensures a *unique* permission.

The SnapTree class, which implements the ITree interface, contains two boolean fields, `isSnapshot` and `hasSnapshot`, and a field `root`, which contains a handle to the root node.

12.3.1 Formula Guarded by a Boolean Variable and Implication

The invariant for *Snapshot* is straightforward. It contains an *immutable* permission to the root in the *PartOfASnapshot* state; the `isSnapshot` field is true, and the `hasSnapshot` field is false.

The field `isSnapshot` is used in the invariant to distinguish between the *Tree* and *Snapshot* states.

For the *Tree* invariant we distinguish between two cases: either there is a snapshot present, or there is no snapshot present. In the former case the invariant contains an *immutable* permission to the root node in the *PartOfASnapshot* state. This ensures the no node is mutated. In the latter case the invariant contains a *unique* permission to the root node in the *NotPartOfASnapshot* state.

To implement this conditional we use a proof pattern: the permission is guarded by an implication whose left hand side tests a boolean program variable. The variable `hasSnapshot` is compared to true (or false), and on the right hand side of the implication we have an *immutable* (or *unique*, respectively) permission to the root node in the *PartOfASnapshot* (or *NotPartOfASnapshot*) state.

```
@ClassStates({
  @State(name="Snapshot", inv="immutable(root) in PartOfASnapshot *
    isSnapshot == true * hasSnapshot == false"),
  @State(name="Tree", inv="isSnapshot == false *
    (hasSnapshot == true => immutable(root) in PartOfASnapshot) *
    (hasSnapshot == false => unique(root) in NotPartOfASnapshot)")
})
```

This distinction between the two cases is natural and follows from the program implementation.

12.3.2 Specification of a Recursive Structure

This implementation either contains a completely immutable tree (if snapshots are present) or a mutable tree. This is specified by the invariants of the states of the node class. Two states are defined, and both refine *alive*: either the node is part of a snapshot (*PartOfASnapshot*) or not part of a snapshot (*NotPartOfASnapshot*). The invariant recursively contains *immutable* (or *unique*) permissions in the *PartOfASnapshot* (or *NotPartOfASnapshot*, respectively) state for the left and right children.

```
@Refine({
  @States(refined="alive",
          value={"PartOfASnapshot","NotPartOfASnapshot"}),
})
@ClassStates({
  @State(name="PartOfASnapshot",
         inv="immutable(left) in PartOfASnapshot *
              immutable(rght) in PartOfASnapshot"),
  @State(name="NotPartOfASnapshot",
         inv="unique(left) in NotPartOfASnapshot *
              unique(rght) in NotPartOfASnapshot")
})
```

The base case for the recursion is that both the left and the right child are null. Plural assumes the possibility that these might be null by default.

12.3.3 Conditional Composition of Implementations

The method `add` behaves differently for a mutable tree and an immutable one. The `add` method in the SnapTree checks in which case the tree is and calls the correct method, either a mutating or a functional insert. In both cases the precondition and postcondition are a *full* permission to an object in the *Tree* state. The annotation `use=Use.FIELDS` specifies that `this` has to be unpacked in the method body, which is required to access the fields.

The implementation first checks whether `root` is null and instantiates a new `Node` object if that is the case. Otherwise the boolean field `hasSnapshot` is tested to determine whether a mutating insert (`addM`)

or a functional insert (addF) should be done. The proof goes through because the test is the same as in the invariant of the *Tree* state, thus one guard is false, its implication is eliminated, and the other guarded formula is used.

```
@Full(use=Use.FIELDS, requires="Tree", ensures="Tree")
public boolean add (int i) {
  assert(isSnapshot == false);
  if (root == null) {
    setRoot(new Node(i));
    return true;
  } else
    if (hasSnapshot) {
      RefBool x = new RefBool();
      setRoot(root.addF(i, x));
      return x.getValue();
    } else {
      RefBool x = new RefBool();
      root.addM(i, x);
      return x.getValue();
    }
}
```

The implementation of addF requires an *immutable* permission of the node in the *PartOfASnapshot* state, and ensures an *immutable* permission in the *PartOfASnapshot* state for the returned object. It recurses down the tree to find the location at which to insert the given value, and if the value was inserted, it duplicates the entire path (which is on the call stack). It uses some helper methods to get and set fields.

```
@Perm(requires="immutable(this) in PartOfASnapshot",
      ensures="immutable(result) in PartOfASnapshot")
public Node addF (int i, RefBool x) {
  Node node = this;
  if (item > i) {
    Node lef = getLeft();
    Node newL = null;
    if (lef == null) {
      newL = new Node(i);
      x.setValue(true);
```

```
    } else
      newL = lef.addF(i, x);
    if (x.getValue()) {
      Node r = getRight();
      node = new Node(newL, item, r);
    }
  } else if (i > item) {
    Node rig = getRight();
    Node newR = null;
    if (rig == null) {
      newR = new Node(i);
      x.setValue(true);
    } else
      newR = rig.addF(i, x);
    if (x.getValue()) {
      Node l = getLeft();
      node = new Node(l, item, newR);
    }
  }
  return node;
}
```

The implementation of addM also searches for the correct place by calling itself recursively, and assigns a freshly instantiated Node object to that place.

```
@Unique(use=Use.DISP_FIELDS,
        requires="NotPartOfASnapshot",
        ensures="NotPartOfASnapshot")
public void addM (int i, RefBool x) {
  if (item > i)
    if (left == null) {
      left = new Node(i);
      x.setValue(true);
    } else
      left.addM(i, x);
  else if (i > item)
    if (rght == null) {
      rght = new Node(i);
      x.setValue(true);
```

```
    } else
      rght.addM(i, x);
}
```

12.3.4 Dropping Privileges (Ghost Method)

The method snapshot requires a *full* permission to this in the *Tree* state. The `!fr` annotation is equivalent to `use=Use.FIELDS`, but can be used in the more general `Perm` annotation.

The implementation of snapshot needs to drop the permissions to all nodes, because they are now shared with the tree and the snapshot. This is achieved in the snapall method.

```
@Perm(requires="full(this!fr) in Tree",
      ensures="pure(result) in Snapshot * full(this!fr) in Tree")
public ITree snapshot() {
  assert(!isSnapshot);
  if (hasSnapshot)
    return new SnapTree(root);
  else {
    Node r = root;
    r.snapall();
    hasSnapshot = true;
    return new SnapTree(r);
  }
}
```

The method snapall drops the privileges recursively by traversing the tree. It is implemented in the Node class. It does not have any observable computational effect, but it is required because we must drop the permissions for the entire tree and Plural only allows this to occur as each node is unpacked going down the tree. In order to verify it with Plural, we need to specifically assign null to the left/right sibling if it is already null (to associate a bottom permission).

```
@Perm(requires="unique(this!fr) in NotPartOfASnapshot",
      ensures="immutable(this!fr) in PartOfASnapshot")
public void snapall () {
```

```
  if (left != null)
    left.snapall();
  else
    left = null;
  if (rght != null)
    rght.snapall();
  else
    rght = null;
}
```

Although the specific technique used here is specialized for Plural, note that an analogous mechanism would be required to convince any tool that the permissions and/or typestates are dropped recursively.

12.3.5 Iterator

The iterator implementation uses a field `context`, which contains a stack of nodes that have not yet been yielded to the client. This is initially filled recursively with the left path, and whenever an item is popped from the stack, the left path of its right subtree is pushed onto the stack. The Stack class is annotated with a proper specification, but its implementation is not verified (especially that `pop` returns an object in the *PartOfASnapshot* typestate)[3].

```
@Perm(requires="immutable(this) in Snapshot",
      ensures="unique(result)")
public TreeIterator iterator() {
  Node r = this.getRoot();
  TreeIteratorImpl it = new TreeIteratorImpl(this);
  it.pushLeftPath(r);
  return it;
}
```

The concrete class specifies an invariant only for the top-level `alive` state:

[3]Update from October 2013: The Stack class, as annotated, does not track whether it is empty or not. A more precise specification could possibly use different typestates carrying this information, which could then be used by the `hasNext` method to specify the connection between the contents of the stack and the typestate of the iterator.

```
@ClassStates({
  @State(name="alive",
         inv="immutable(tree) in Snapshot *
              unique(context) in alive")
})
```

The method pushLeftPath calls itself recursively with the left child to push the entire left path onto the stack. It requires a *unique* permission to this in order to unpack this, access the context field, and call a method on the context object while this remains unpacked. As mentioned in Section 12.1, for soundness reasons, leaving an object unpacked during a method call is only possible in Plural if there is a *unique* permission to the unpacked object.

```
@Perm(requires="unique(this!fr) in alive *
                immutable(#0) in PartOfASnapshot",
      ensures="unique(this!fr) in alive *
               immutable(#0) in PartOfASnapshot")
public void pushLeftPath(Node node) {
  if (node != null) {
    context.push(node);
    pushLeftPath(node.getLeft());
  }
}
```

The hasNext method is simply a check whether the stack is non-empty[4].

```
@TrueIndicates("NonEmpty")
@FalseIndicates("Empty")
@Imm(use=Use.FIELDS)
public boolean hasNext() {
  return !context.empty();
}
```

[4]Update from October 2013: The verification of the hasNext method is very lightweight. Especially it does not verify whether the stack (context) is actually empty or not, since the stack specification does not expose this information via a typestate (also see footnote on page 174).

The method next pops the first element of the stack and pushes the left path of the right child onto the stack. In contrast to the original implementation [82], a guard if hasNext() is true around lines 4-7 is not needed, because Plural verifies that next is only called on a non-empty iterator.

```
@Unique(use=Use.DISP_FIELDS, requires="NonEmpty")
public Integer next() {
3   Integer result;
    Node node = context.pop();
    result = node.getItem();
6   if (node.getRight() != null)
      pushLeftPath(node.getRight());
    return result;
9 }
```

Here a *unique* permission is required in order to call pushLeftPath.

In this section we described proof patterns used in the verification of the path copy persistence implementation of snapshotable trees. The complete implementation has been automatically verified with Plural.

12.4 Related Work

The Composite pattern, which is a tree data structure, has been verified using typestate and access permissions [19]. This work differed in multiple aspects: first of all it was not formalized in a tool, then it relied on extensions, like multiple unpacking and equations using pointers, which were not proven to be sound. Also, the verification challenge is different: the Composite pattern exposes all nodes to a user using a *share* permission, and preserves an invariant upwards the tree, namely the number of children of the subtree rooted in each node. This leads to a specification with several typestates in the different dimensions of each node, which fractions are cleverly distributed to allow for bottom-up updates of the count.

A prior iterator verification [16] is similar to our specification, but the implementation of the iterator is completely different. In this paper we present an iterator which shares its content with the snapshot and

holds only some elements on the stack, pushing more onto the stack on demand.

Snapshotable trees have been verified using a higher-order separation logic [82]. This approach verified full functional correctness, while this paper can only prove correct API usage: `add` and `snapshot` are always called on the tree, and by having *immutable* permission to the contents of a snapshot, we can verify that it will not be modified. Also, our work verifies that an iterator is always taken on a snapshot, not the original tree, and that `next` is never called on an empty iterator.

We use automation in the proof, which requires only a moderate number of annotations to the source code. The higher-order separation logic proof requires roughly 5000 lines of proof script, while the code and annotations for this paper are together under 400 lines; this is less than 2 lines of annotation for every line of source code.

An unpublished verification of snapshotable trees in Dafny [74], done by Rustan Leino, is similar to the Plural approach. Both are automated systems using a first-order logic. In Dafny functional correctness can be proven. The advantage of Plural is that already existing code written in a widely deployed programming language (Java) can be analyzed, whereas Dafny specifies its own programming language. Dafny uses implicit framing and also relies on annotations by the user, whereas Plural is based on linear logic (access permissions) and typestates. Dafny does not support inheritance, thus no abstract specification is provided.

12.5 Conclusion and Further Work

There exist several extensions to the access permission system which support verifying full functional correctness: Object propositions [91] combine access permissions with first-order formulae; but there is currently no implementation available. Symplar [18] combines access permissions with JML, thus access permissions are used to reason about aliasing, and JML formulae for full functional correctness.

In order to verify iterators over the tree (vs. its snapshots) we would need to change the *unique* permission of the nodes to *full* in order to share them between the tree and the iterator. Because the proof relies

on method calls while a *unique* object is unpacked, we would have to modify Plural in order to achieve this.

There are also more advanced implementations of snapshotable trees [46], namely rebalancing - for which we would need to have partly *unique* and partly *immutable* permissions to the nodes in the tree. An important observation is that rebalancing involves only freshly allocated nodes in the path copy persistence implementation. Thus, we would need to carefully write the code such that Plural can derive this observation.

The node copy persistence implementation is more challenging: parts of a node are immutable while other parts are mutable. Here orthogonal dimensions of state, which are implemented in Plural, might become useful.

To conclude this paper, we successfully verified a snapshotable tree implementation and client code in Plural. In order to achieve that we had to rewrite parts of the reference implementation [82], mainly by adding explicit getter and setter methods, which is good object-oriented style.

An interesting method was `add`, which in the reference implementation calls `addRecursive`, which handles all cases at once: whether a snapshot is present (functional insertion) or no snapshots are present (mutating insert). In the higher-order separation logic proof this leads to three different specifications for `addRecursive`, one for each separate case. In automated tools (Plural and Dafny), it is easier to implement and verify two methods for those two cases, due to size of invariants and automated reasoning. Evidence for this is also provided by Rustan Leino, who implemented insertion in a clean-room setting from the beginning as two different methods. The reference implementation is clearly more compact, but it is arguable which implementation is clearer or more in the object-oriented spirit.

We modified the client code slightly by removing an additional temporary boolean variable, because we found that Plural's inference of boolean values works better this way. The original challenge used a boolean variable because their semantics does not allow for statements (heap access) in the loop condition, but only expressions (stack access).

While doing this proof we found several proof patterns for Plural: using implications instead of multiple typestates, inserting explicit re-

turn statements to help Plural with automation, writing explicit alternatives for conditionals, moving methods into the specific class that concerns them because static methods are not as well supported, avoiding choice conjuncts, and assigning null explicitly so that Plural can associate a bottom permission with the field. To get the proof through, we had to write the method `snapall`, which does not have any observable computational effect, but reassigns fields which were null to null.

We consider Plural to be a helpful static analysis tool which prevents runtime bugs: it issues an error when `add` is called on a snapshot or when a snapshot of a snapshot is taken.

One bug in Plural has been found (`while (lc == true)` leads to infinite recursion), which silently crashed Plural, making it appear that the code was proven. This has subsequently been fixed by the author of Plural.

Many thanks to Kevin Bierhoff for helping with specifications and best practices in Plural. The second author was funded by NSF grant CCF-1116907.

Bibliography

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] A. Andersson. Balanced search trees made simple. In F. Dehne et al., editors, *Algorithms and Data Structures. LNCS 709*, pages 60–71. Springer-Verlag, 1993.

[3] A. W. Appel. Tactics for separation logic. *INRIA Rocquencourt and Princeton University, Early Draft*, 2006.

[4] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 109–122, New York, NY, USA, 2007. ACM.

[5] D. Aspinall. Proof General: A Generic Tool for Proof Development. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000.

[6] H. G. Baker. "Use-once" variables and linear objects: storage management, reflection and multi-threading. *SIGPLAN Not.*, 30:45–52, January 1995.

[7] M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. In *VSTTE*, pages 144–152, 2005.

[8] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK - a tool for validation of security and behaviour of Java applications. In *FMCO*, pages 152–174, 2006.

[9] G. Barthe, P. Crégut, B. Grégoire, T. Jensen, and D. Pichardie. The MOBIUS proof carrying code infrastructure. In F. S. Boer, M. M. Bonsangue, S. Graf, and W.-P. Roever, editors, *Formal Methods for Components and Objects*. Springer Verlag, 2008.

[10] N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *ECOOP'11*, 2011.

[11] J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! – a framework for higher-order separation logic in Coq. In *ITP*, pages 315–331, 2012.

[12] J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *ITP*, pages 22–38, 2011.

[13] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2006.

[14] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[15] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: the Calculus of Inductive Constructions*. Springer Verlag, 2004.

[16] K. Bierhoff. Iterator specification with typestates. In *Proceedings of the 2006 conference on Specification and verification of component-based systems*, SAVCBS '06, pages 79–82, New York, NY, USA, 2006. ACM.

[17] K. Bierhoff. API protocol compliance in object-oriented software. Technical Report CMU-ISR-09-108, CMU ISR SCS, 2009.

[18] K. Bierhoff. Automated program verification made SYMPLAR. In *Proc of Onward! 2011*, 2011.

[19] K. Bierhoff and J. Aldrich. Permissions to specify the composite design pattern. In *Proc of SAVCBS 2008*, 2008.

[20] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[21] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[22] J. M. Bland and D. G. Altman. Statistics notes: Cronbach's alpha. *BMJ*, 314(7080):572, 2 1997.

[23] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[24] R. L. Bocchino, H. Mehnert, and J. Aldrich. High-level abstractions for safe parallelism. In *4th Workshop on Determinism and Correctness in Parallel Programming*, 2013.

[25] C. Borrás. Overexposure of radiation therapy patients in Panama: problem recognition and follow-up measures. *Rec Panam Salud Publica*, 20(2/3):173–187, 2006.

[26] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[27] E. Brady. Programming in Idris: a tutorial. Technical report, University of St Andrews, 2013.

[28] F. P. Brooks. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7, Jun 2005.

[30] C. Calcagno and D. Distefano. Infer: an automatic program verifier for memory safety of c programs. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 459–465, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Jan 2009.

[32] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Space invading systems code. In *LOPSTR*, pages 1–3, 2008.

[33] J. Charles and J. R. Kiniry. A lightweight theorem prover interface for Eclipse. *UITP at TPHOL'08*, 2008.

[34] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011.

[35] A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. *ACM Proc. of ICFP '09*, Aug 2009.

[36] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML – progress and issues in building and using ESC/Java2. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop*. Springer, 2004.

[37] L. J. Cronbach. Coefficient alpha and the internal structure of tests. *Psychometrika*, 16:297–334, 1951.

[38] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., London, UK, UK, 1972.

[39] C. David and W.-N. Chin. Immutable specifications for more concise and precise verification. In C. V. Lopes and K. Fisher, editors, *OOPSLA*, pages 359–374. ACM, 2011.

[40] F. Davis. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *Management Information Systems Quarterly*, 1989.

[41] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[42] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In T. D'Hondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 504–528. Springer Berlin / Heidelberg, 2010.

[43] T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstraction and refinement for local reasoning. In *VSTTE*, pages 199–215, Berlin, Heidelberg, 2010. Springer.

[44] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer Berlin Heidelberg, 2006.

[45] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In G. E. Harris, editor, *OOPSLA*, pages 213–226. ACM, 2008.

[46] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86 – 124, 1989.

[47] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, May 1986.

[48] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. *ACM Proc. of SAC '10*, Mar 2010.

[49] J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. *Proc. of CAV'07*, Jul 2007.

[50] G. Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *ASCM*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[51] G. Gonthier. Engineering mathematics: the odd order theorem proof. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 1–2. ACM, 2013.

[52] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[53] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th FCS, Ann Arbor, Michigan*, pages 8–21, 1978.

[54] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data structure fusion. In K. Ueda, editor, *APLAS*, volume 6461 of *Lecture Notes in Computer Science*, pages 204–221. Springer, 2010.

[55] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[56] W. A. Howard. The Formulae-as-types Notion of Construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980. original manuscript from 1969.

[57] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[58] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, pages 41–55, 2011.

[59] J. B. Jensen and L. Birkedal. Fictional separation logic. In H. Seidl, editor, *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 377–396. Springer, 2012.

[60] J. B. Jensen, L. Birkedal, and P. Sestoft. Modular verification of linked lists with views via separation logic. *Proc. of FTfJP 2010*, May 2010.

[61] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University, November 1992.

[62] I. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin Heidelberg, 2006.

[63] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[64] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *SOSP*, pages 207–220. ACM, 2009.

[65] D. E. Knuth. Literate programming. *THE COMPUTER JOURNAL*, 27:97–111, 1984.

[66] N. Kokholm and P. Sestoft. The C5 Generic Collection Library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, January 2006.

[67] N. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, June 2012.

[68] N. R. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *TLDI*, pages 63–76. ACM, 2010.

[69] N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *Proceedings of the 17th annual ACM SIGPLAN-SIGACT International Conference on Functional Programming*, ICFP '12, New York, NY, USA, September 2012. ACM.

[70] V. Kuncak and M. C. Rinard. An overview of the Jahob analysis system: project goals and current status. In *IPDPS*. IEEE, 2006.

[71] O. Laitenberger and H. M. Dreyer. Evaluating the Usefulness and the Ease of Use of a Web-based Inspection Data Collection Tool. In *IEEE International Software Metrics Symposium*, pages 122–132, 1998.

[72] O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 592–608, Utah, USA, July 2011. Springer-Verlag.

[73] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.

[74] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning. LNCS 6355*, pages 348–370, 2010.

[75] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[76] N. G. Leveson. An investigation of the Therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[77] B. Liskov and J. Wing. A behavioral notion of subtyping. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(6), Nov 1994.

[78] G. Malecha and G. Morrisett. Mechanized verification with sharing. In *7th International Colloquium on Theoretical Aspects of Computing*, Sept. 2010.

[79] H. Mehnert. Kopitiam: Modular incremental interactive full functional static verification of Java code. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 518–524. Springer Berlin Heidelberg, 2011. Reproduced as Chapter 6 of this dissertation.

[80] H. Mehnert and J. Aldrich. Verification of snapshotable trees using access permissions and typestate. In C. A. Furia and S. Nanz, editors, *TOOLS (50)*, volume 7304 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2012. Reproduced as Chapter 12 of this dissertation.

[81] H. Mehnert and J. Bengtson. Kopitiam – a unified IDE for developing formally verified Java programs. Technical Report ITU-TR-2013-167, IT University of Copenhagen, May 2013. Reproduced as Chapter 7 of this dissertation.

[82] H. Mehnert, F. Sieczkowski, L. Birkedal, and P. Sestoft. Formalized verification of snapshotable trees: Separation and sharing. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 179–195. Springer Berlin Heidelberg, 2012. Reproduced as Chapter 10 of this dissertation.

[83] B. Meyer. Design by contract. *Advances in Object-Oriented Software Engineering*, 1991.

[84] W. Mostowski. Fully verified Java card API reference implementation. In *VERIFY*, 2007.

[85] L. D. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[86] K. Mulmuley. A fast planar partition algorithm, I. *Journal of Symbolic Computation*, 10(3-4):253 – 280, 1990.

[87] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In J. Hook and P. Thiemann, editors, *Proc. of 13th ACM ICFP 2008*, pages 229–240. ACM, 2008.

[88] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of POPL*, 2010.

[89] J. Narboux. A graphical user interface for formal proofs in geometry. *Journal of Automated Reasoning*, 39(2):161–180, 2007.

[90] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[91] L. Nistor and J. Aldrich. Verifying object-oriented code using object propositions. In *Proc of IWACO*, 2011.

[92] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[93] M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *Programming Languages and Systems*, volume 6602 of *Lecture Notes in Computer Science*, pages 439–458. Springer Berlin Heidelberg, 2011.

[94] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *POPL*, pages 247–258. ACM, 2005.

[95] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In G. C. Necula and P. Wadler, editors, *POPL*, pages 75–86. ACM, 2008.

[96] R. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative Hoare type theory. In S. Drossopoulou, editor, *ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2008.

[97] P. Philippaerts, F. Vogels, J. Smans, B. Jacobs, and F. Piessens. The Belgian electronic identity card: a verification case study. *ECEASST*, 46, 2011.

[98] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[99] B. C. Pierce, C. Casinghino, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2012. `http://www.cis.upenn.edu/~bcpierce/sf`.

[100] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *IEEE Proc. of 17th Symp. on Logic in CS*, pages 55–74, Nov 2002.

[101] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of ACM*, 29(7):669–679, July 1986.

[102] N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *LPAR*, pages 398–414, 2005.

[103] P. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering, 14th IEEE International Conference*, pages 139–148, 2006.

[104] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science*, 7(3:21), July 2011.

[105] R. Sedgewick. Left-leaning red-black trees. At http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf.

[106] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. *ECOOP 2009*, Apr 2009.

[107] G. Stewart, L. Beringer, and A. W. Appel. Verified heap theorem prover by paramodulation. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 3–14, New York, NY, USA, 2012. ACM.

[108] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. In *IEEE Transactions on Software Engineering*, 1998.

[109] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and Éric Tanter. First-class state change in Plaid. In *OOPSLA'11*, 2011.

[110] K. Svendsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates. In *ECOOP'10*, pages 175–199. Springer-Verlag, 2010.

[111] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.3*, May 2002.

[112] R. Vallée-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical Report 4, McGill University, 1998.

[113] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *TACAS*, pages 299–312, 2001.

[114] C. Varming and L. Birkedal. Higher-Order Separation Logic in Isabelle/HOLCF. *Electronic Notes in Theoretical Computer Science*, 218:371–389, 2008.

[115] F. Vogels, B. Jacobs, and F. Piessens. A machine-checked soundness proof for an efficient verification condition generator. In S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, editors, *SAC*, pages 2517–2522. ACM, 2010.

[116] B. Weide, M. Sitaraman, H. Harton, B. Adcock, P. Bucci, D. Bronish, W. Heym, J. Kirschenbaum, and D. Frazier. Incremental benchmarks for software verification tools and techniques. *Proc. of VSTTE '08*, Oct 2008.

[117] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *CADE*, pages 140–145, 2009.

[118] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok. Recent catastrophic accidents: Investigating how software was responsible. In *SSIRI*, pages 14–22. IEEE Computer Society, 2010.